Intermediate CONNECT Architecture

Amel Bennaceur, Gordon S. Blair, Franck Chauvel, Nikolaos Georgantas, Paul Grace, Valérie Issarny, Vatsala Nundloll, Massimo Paolucci, Rachid Saadi, Daniel Sykes

CONNECT

Emergent Connectors for Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D1.2

Intermediate CONNECT Architecture

http://www.connect-forever.eu

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	Report

Deliverable Number	:	D1.2
Title of Deliverable	:	Intermediate CONNECT Architecture
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	24
Contractual Delivery Date	:	1st February 2011
Actual Delivery Date	:	18th February 2011
Contributing WPs	:	WP1
Editor(s)	:	Paul Grace, Gordon S. Blair (LANCS)
Author(s)	:	Amel Bennaceur (INRIA), Gordon S. Blair (LANCS), Franck Chauvel (PKU), Nikolaos Georgantas (INRIA), Paul Grace (LANCS), Valerie Issarny (INRIA), Vatsala Nundloll (LANCS), Massimo Paolucci (DOCOMO), Rachid Saadi (INRIA), Daniel Sykes (INRIA)
Reviewer(s)	:	CNR

Abstract

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. This challenge is exasperated by the highly heterogeneous technologies employed by each of the interacting parties, i.e., in terms of hardware, operating system, middleware protocols, and application protocols. The key aim of the CONNECT *project is to drop this heterogeneity barrier and achieve universal interoperability. Here we report on the activities of WP1 into developing the* CONNECT *architecture that will underpin this solution. In this respect, we present the following key contributions from the second year. Firstly, the intermediary* CONNECT *architecture that presents a more concrete view of the technologies and principles employed to enable interoperability between heterogeneous networked systems. Secondly, the design and implementation of the discovery enabler with emphasis on the approaches taken to match compatible networked systems. Thirdly, the realisation of* CONNECT*ors that can be deployed in the environment; we provide domain specific language solutions to generate and translate between middleware protocols. Fourthly, we highlight the role of ontologies within* CONNECT *and demonstrate how ontologies crosscut all functionality within the* CONNECT *architecture.*

Keyword List

Interoperability, middleware, middleware heterogeneity, service discovery heterogeneity, service interaction heterogeneity, application-level heterogeneity, data heterogeneity, non-functional properties, software architecture, connectors, semantics, ontologies.

Document History

Version	Type of Change	Author(s)
1	Document creation	Paul Grace
2	Added complete Chapter 1 (Introduction)	Paul Grace
3	Added Initial Content for Chapter 3	Daniel Sykes
4	Added Initial Content for Chapters 2 and 5	Paul Grace
5	Added Initial Content for Chapter 6	Vatsala Nundloll
6	Added Section 7.1 to Chapter 7	Paul Grace
7	Revisions and new content to Chapter 3	Daniel Sykes
8	Added Content for Sections 2.1 and 2.2	Paul Grace
9	Revisions and new content for Chapter 6	Massimo Paolucci
10	Revised Chapter 2 and added Content for Section 4	Paul Grace
11	Added Initial Content for Chapter 5	Franck Chauvel
12	Revisions to Chapter 3	Daniel Sykes
13	Added content for sections 4.3.1 and 4.3.2	Franck Chauvel
14	Added section 6.3	Massimo Paolucci
15	Added chapters 8 and 9	Paul Grace
16	Revisions to document (Internal Draft Version)	Paul Grace
17	Restructure of Chapter 6	Massimo Paolucci
18	Added content to section 3.2	Amel Bennaceur
19	Added content for Chapter 8	Paul Grace
20	Revisions to chapter 3 and 5	Amel Bennaceur, Rachid Saadi
21	Added abstract and revised chapter 1 to follow common format for all deliverables	Paul Grace
22	Completed content for chapter 3	Amel Bennaceur, Rachid Saadi
23	Revised version according to reviews	Paul Grace, Massimo Paolucci
24	Final version	

Document Review

Date	Version	Reviewer	Comment
28th January 2011	21	Illaria Matteucci (CNR)	1) In the introduction identify that the document focuses on discovery and synthesis. Explain why non-functional properties are not addressed here. 2. Fig 2.10 doesn't include the trust and security enablers. Potentially complete the figure and say that during this year we focus on the functionalities and the interactions of some enablers only. 3. Why is there a chapter about synthesis? What are the links with the theory presented in WP3? These links should be made explicit. The chapter about synthesis may conflict (and confuse) with deliverable D3.2. 4. The concepts of mediators and CONNECTors have to be made homogeneous throughout the deliverables. 5. The chapter on ontology is heavily weighted towards VANET protocols, and would be better highlighting a range of CONNECT application domains. 6. Minor corrections of text and presentation.

Table of Contents

List of Acronyms

API	Application Programming Interface
BPEL	Business Process Execution Language
CADL	CONNECT Action Definition Language
CDP	CONNECT Dicovery Protocol
CSMD	Client/Service and Message-oriented Description
DPWS	Devices Profile For Web Services
DSL	Domain Specific Language
FSP	Finite State Processes
GIOP	General Inter-ORB Protocol
IIOP	Internet Inter-ORB Protocol
IM	Instant Messaging
IP	The Internet Protocol
JMS	Java Messaging Service
LIME	Linda in a Mobile Environment
LTSA	Labelled Transition System Analyzer
MANET	Mobile Ad hoc Network
MDL	Message Description Language
MSMQ	Microsoft Message Queue
MSNP	MSN Messenger protocol
NFP	Non-functional properties
NSDL	CONNECT Networked System Description Language
OFSP	Ontological Finite State Processes
OLTSA	Ontological Labelled Transition System Analyzer
OWL	Ontology Web Language
PSD	Publish/Subscribe Description
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAWSDL	Semantic Annotations for Web Services Description Language
SDP	Service Discovery Protocol
SLP	Service Location Protocol
SMD	Shared Memory Description
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UPnP	Universal Plug-And-Play
VANET	Vehicular Ad hoc Network
WSDL	Web Services Description Language
xDL	extensible Description Language
XMPP	Extensible Messaging and Presence Protocol
YMSG	Yahoo! Messenger Protocol

List of Figures

1 Introduction

The CONNECT project is investigating a fundamental requirement of pervasive computing systems; that systems developed independently from one another must be able to *interoperate*. Interoperability is defined as the ability of two systems to exchange, understand and use data from one another. Importantly, pervasive computing is typically composed of systems and applications that are highly heterogeneous in terms of the software and protocols employed. In the previous Deliverable D1.1 [6] we identified five types of heterogeneity in such systems that are a barrier to achieving interoperability:

- *Discovery protocol heterogeneity*. Different protocols are used to advertise and search for services, e.g., Service Location Protocol (SLP), Jini, Universal Plug and Play (UPnP), and Lightweight Directory Access Protocol (LDAP).
- *Interaction protocol heterogeneity*. Services use different protocols to exchange and use data, e.g., Remote Method Invocation protocols such as SOAP, Java RMI and IIOP; or different messaging protocols such as Java Message Service (JMS) or Microsoft Message Queuing (MSMQ).
- *Data heterogeneity*. Applications may use data that is represented in different ways and/or have different meanings.
- *Application heterogeneity*. The application interfaces may be different in terms of the descriptions of operations, e.g., the behaviour provided by one operation in one interface may be provided by multiple operations in the other interface. Interfaces are also heterogeneous in terms of the order in which operations must/should be called.
- *Heterogeneity of Non-functional properties*. Systems may have particular non-functional properties, e.g., latency of message delivery, dependability measures and security requirements that must be resolved with respect to the connected system.

Our survey of existing academic/research and industrial solutions [6] showed that no solution from either the middleware or semantic interoperability fields addressed all of these heterogeneity dimensions. Hence, CONNECT aims to go beyond these approaches and resolve the interoperability challenge in a fundamentally different way; the behaviour of networked systems is discovered, monitored and learned and based upon this a CONNECTor is dynamically synthesized that will ensure two systems will interoperate.

1.1 The Role of Work Package WP1

The aim of WP1 is to provide an overall architecture for CONNECT, defining and documenting the common architectural principles behind the CONNECT approaches to achieving extremely long-lived (eternal) networked systems. The original three tasks of WP1 as described in the description of work [13] are as follows:

Task 1.1: CONNECT architecture. Elaborating a technology-independent and eternal architectural framework for emergent CONNECTors.

Task 1.2: Eternal system semantics. Eliciting an ontology-based characterization of the semantics of connected systems.

Task 1.3: CONNECT realization. Developing key underlying systems principles and techniques to support the development of a practical, efficient and a self-sustaining CONNECT prototype.

Hence, this work package performs a central role for CONNECT as a whole: acting as a point of integration for the specialized work from each of the work packages [14] [18] [17] [15]; and providing the system prototypes to directly support the experimentation and evaluation work of the project.

1.2 Summary of Achievements in Year One: The Initial CONNECT Architecture

In the first year of the project, the following key contributions where achieved [6]:

- Based upon analysis of typical complex pervasive systems, we identified the five important dimensions of heterogeneity (described previously) that are encountered and must be resolved by CONNECT.

- We produced a state of the art of middleware and data interoperability solutions; this showed that no solution achieves the interoperability proposed by CONNECT; and hence highlighted the key contributions that this project can make to the field.

- A description of the initial technology-independent and eternal architectural framework and architectural principles was presented; this was validated against a small number of case studies.

1.3 Challenges for Year Two

This initial architecture was the first step on the road to achieving future proof interoperability; and with this first step there were naturally a number of open questions raised with corresponding recommendations by the project reviewers.

Becoming Concrete. The initial architecture was deliberately abstract, in order to allow us to refine the architectural principles without being tied to any earlier defined assumptions about networked systems and interoperability. An important concern is: how is the intent of networked systems expressed? Here we document the concrete details of the *Networked System Model* and emphasize its central role within the CONNECT architecture; it is discovered/produced by the discovery and learning enablers and then used by the synthesis and dependability enablers to synthesize CONNECTors. Another challenge related to the integration of enablers, concerns the identification of the specific information they exchange and process. The enabler architecture is presented in this document, focusing solely on the behaviour in terms of the concrete information received and produced by each enabler (the details of the functionality of the enablers is provided in the specialised deliverables).

The Role of Ontologies A further important challenge is to define the role of ontologies within the CONNECT architecture. Interoperability cannot be achieved without semantic matching and mapping of information from one networked system to another. The initial CONNECT architecture was not specific about how ontologies are to be used within the functionality of CONNECT. Here, we emphasize further the important role of ontologies, and describe how they cross-cut the CONNECT architecture: at a high-level (application-level) ontologies are used within the discovery and synthesis enablers to match equivalent networked systems and then produce a mapping from one system to another; at lower-levels (i.e., the middleware/protocol level) ontologies classify new protocol behaviour (e.g., this is an RPC protocol message) and also seek bridges from one middleware protocol to another.

1.4 Achievements in Year Two

In the previous deliverable [6], the initial version of the CONNECT architecture was presented. The objective of this report is to provide a refined version of this architecture, and in particular illustrate both the concrete design and the initial implementation of prototype software that composes the architecture. In particular, we concentrate here on the core functionalities of the architecture: discovery, learning and synthesis; these provide the building blocks of the architecture, which will then be extended with functionality to handle non-functional properties in the third year of the project.

In this report we highlight the following key contributions:

- The CONNECT architecture is presented in Section 2. This focuses on three important elements: i) the CONNECT *Networked System Model*, which offers a rich semantic description of individual systems in terms of their role, interface syntax, behaviour and non-function properties; ii) the *Enabler Architecture*, which integrates the individual enablers that carry out particular roles within the CONNECT process; and iii) the CONNECTor architecture that describes how the software to connect two networked system is constructed.

- The *Discovery Enabler* is presented in Section 3 along with the associated prototype implementation. This enabler forms the important role of initially discovering networked services that are advertised using heterogeneous discovery technologies. A case study is presented to highlight the operation of the discovery enabler and illustrate the features of the software prototype.

- The realisation of CONNECTors is presented in Section 4; we document how concrete communication protocols are implemented within the CONNECTor architecture. An important feature is the use of high-level models (domain specific languages that describe middleware protocols) to generate the required middleware code; this allows CONNECT to be easily extensible for future middleware protocols. Two case studies demonstrate how the realised CONNECTors successfully bridge application and middleware protocols.
- The Deployment enabler is documented in Section 5; here we describe how the CONNECTors are deployed in the environment in order to connect the systems.
- The role of ontologies within the CONNECT architecture is discussed in Section 6. This investigates how ontologies cross-cut the architecture, and in particular how they are specified, also how they are utilised by the individual enablers. Two case studies involving communication protocols (vehicular ad-hoc networks) and system architectures are used to highlight the application of ontologies.

2 Intermediary Connect Architecture

2.1 Introduction

The CONNECT Architecture defines the underlying architectural principles that underpin the work of each of the specialised work packages. The objective is to integrate this work in order to deploy long-lived, universal CONNECTors that resolve interoperability problems between networked systems. The description and definition of the CONNECT Architecture is a living document that is continuously refined during the lifetime of the CONNECT project. In this section, we highlight the enhancements that have been made during the second year. To do this, we first provide a short reminder of the key elements that were described in the Initial CONNECT Architecture [6] at the end of the first year. Subsequently, we provide a roadmap of the refinements that have been made during the second year. These refinements document the key contributions of WP1 in the second year, and point the reader towards more detailed coverage of these contributions in the later chapters of this report, and indeed the other deliverables.

2.2 Overview of the Initial Connect Architecture

In the initial version of the CONNECT architecture [6], we opted for an approach that made the fewest possible assumptions and reflected the CONNECT vision as identified in the project's Description of Work [13]. In this section we shortly summarise the key elements of this version, which continue to form the architectural principles, before then discussing the refinements we have made to this architecture in Section 2.3; the objective of these refinements is to make concrete the earlier abstract concepts and features.

2.2.1 CONNECT Actors

The key actors involved in the CONNECT process were identified as follows (these remain central to the continued refinement of the architecture):

- *Networked systems* are systems that manifest the will to connect to other systems for fulfilling some intent identified by their users and the applications executing upon them.
- *Enablers* are networked entities in the environment of networked systems that incorporate all the intelligence and logic offered by CONNECT for enabling connection between heterogeneous networked systems. Enablers constitute the CONNECT enabling architecture.
- CONNECTors are the emergent connectors produced by the action of enablers.
- CONNECTed systems are the outcome of the successful creation and deployment of CONNECTors.

A high-level view of these actors is shown in Figure 2.1. It can be seen that networked systems manifest their will to connect. This will, along with information about the networked systems, is communicated in the form of some input to the enablers. One or more enablers collaborate to synthesize and deploy a CONNECTor that enables networked systems to connect and fulfill their individual intents.

2.2.2 Networked System Model

The original Networked System Model as seen in Figure 2.2 aimed to model the external interaction behaviour of a networked system. For this, two levels of interaction were considered: middleware-layer interaction and application-layer interaction. The application component describes: an intent, what external behaviour it requires, and what external behaviour it provides. The essential feature here is the interface, that is, a description of the set of functionalities of the component made accessible to (but also required from) its environment. Typically, this description comes in the form of a set of data inputs and associated outputs following a specific data type system.

Regarding middleware-layer interaction, we identified both the provided (by the system) and the required (from other systems) behaviour. Such behaviour was characterized by the following features:

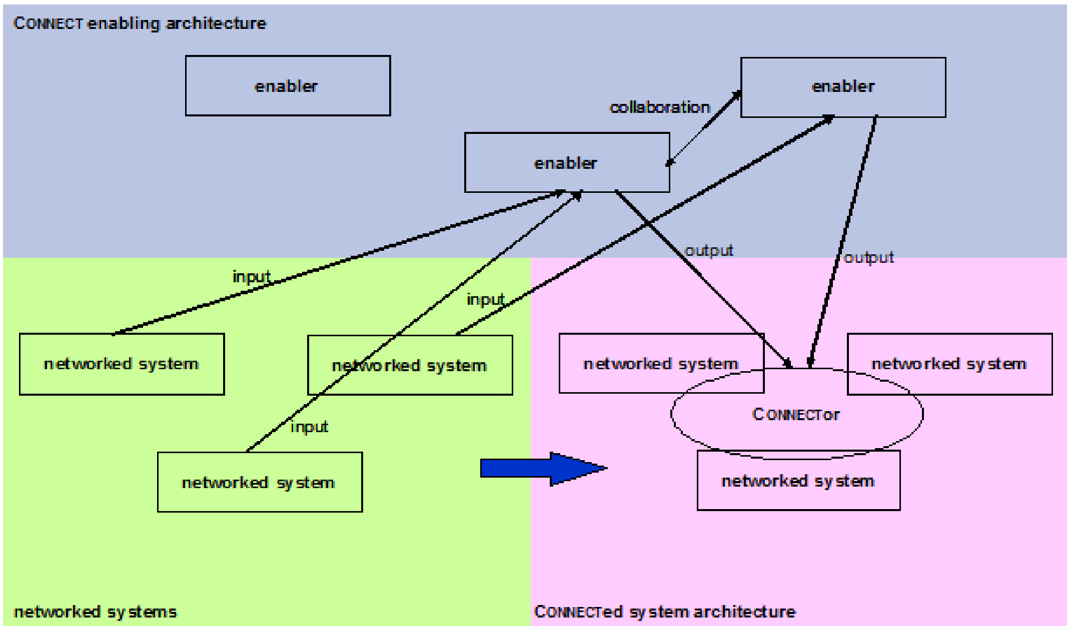


Figure 2.1: Actors in the CONNECT architecture

- Process specifies the supported interaction protocol, e.g., in the form of sequences of exchanged messages and states.
- Coordination patterns characterize the role semantics in an interaction, such as client-server or peer-to-peer, orchestration or choreography.
- Interaction patterns characterize the semantics of interaction protocols, such as message-based, event-based or shared-memory-based, synchronous or asynchronous.
- Data transfer protocol specifies the representation of data and control in the messages conveyed by the interaction protocol.
- The Addressing scheme specifies the naming and referencing convention employed to uniquely identify a networked entity.
- The Data type system specifies the representation of data types applied to all the data conveyed by the data transfer protocol.
- Data is the actual data information conveyed by the data transfer protocol.

Regarding application-layer interaction, we identified a subset of the features described above for the middleware-layer interaction. These features have similar meanings at the application layer. In general, applications rely on middleware for their external interaction, which means that they incorporate the semantics of the underlying middleware, and add their own semantics on top of that. For example, a process specifies the logic of an application more or less independently of the underlying middleware, and an application that uses shared memory can implement many different coordination and interaction patterns on top of this middleware. In addition to the functional features, the networked system model aims to include non-functional properties of networked systems, which are orthogonal to the functional aspects. In this view, we considered provided and required non-functional properties, their description, and their enforcement, that is, how they are functionally implemented by the networked system.

2.2.3 Phases of the CONNECT Runtime

In accordance with the overall view of the CONNECT open environment and architecture we identified the following phases of the CONNECT runtime:

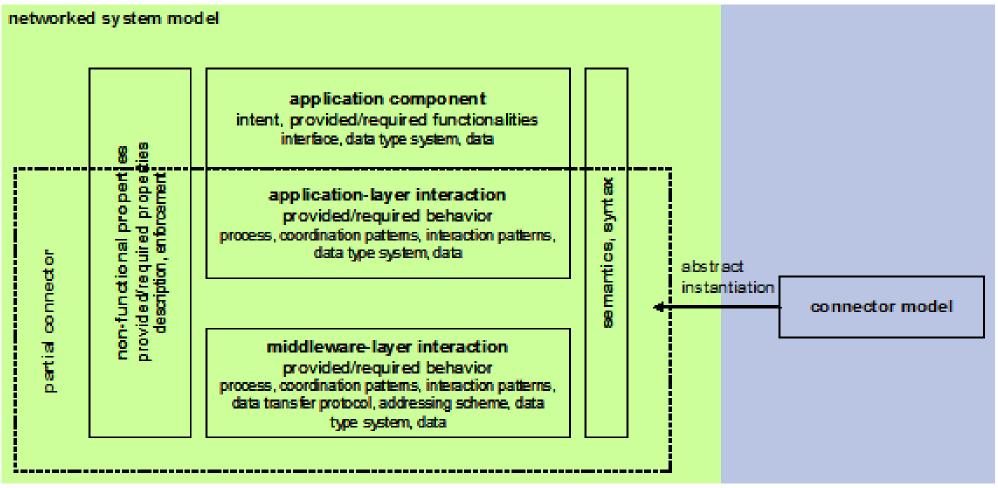


Figure 2.2: The original CONNECT Networked System Model

- Discovery enables networked systems to manifest their will to connect to other networked systems and to discover mutually interested networked systems, while at the same time allows the CONNECT enabling architecture to retrieve initial information on likely-to-be-associated networked systems.
- Learning is performed by enablers upon networked systems for completing the initial information about the latter provided by discovery. The outcome of combined discovery and learning should be a sufficiently good Networked System Model of a networked system.
- Synthesis is performed by enablers for generating and deploying an appropriate CONNECTor that will successfully bridge the heterogeneous systems and establish a CONNECTed system.
- Verification & validation is performed by enablers during and after the synthesis phase for ensuring the correctness of the CONNECTor and the running CONNECTed system with respect to the requirements and intents of the involved networked systems.

These phases composed the initial CONNECT enabling architecture. The behaviour of a typical life cycle pass through this architecture (when connecting two systems) is illustrated in Figure 2.3. Here, triggered by one or more networked systems manifesting their will to connect, interoperable discovery and matching identifies the networked systems that are likely to be associated based on their a priori descriptions, and communicates this information to learning (Step 1). Learning infers the interaction behaviour of the identified networked systems and completes their a priori descriptions, thus eliciting - as precisely as possible - their Networked System Models, which it feeds into synthesis (Step 2). Synthesis generates a CONNECTor model able to bridge the heterogeneous systems (Step 3). By successive model-to-model transformations (Step 4), synthesis generates appropriate models of the CONNECTor required by verification & validation (V&V) (Step 5). V&V evaluates the CONNECTor offline and provides feedback to synthesis, which may lead to reconfiguration or resynthesis of the CONNECTor model (Step 6). At the end of this synthesis and evaluation cycle, synthesis performs a model-to-code transformation to generate an executable CONNECTor (Step 7). Synthesis then deploys the CONNECTor code accomplishing a CONNECTed system, which executes; during this execution, the CONNECTor is able to self-reconfigure to respond to changes in its environment (Step 8). All along the CONNECTed system execution, V&V monitors and evaluates it (including networked systems and CONNECTor) online (Steps 9, 10). At the same time, learning also monitors the CONNECTed system to update and improve its learned models of the constituent networked systems (specifically, V&V and learning apply similar monitoring and model-based testing mechanisms) (Step 11). An evaluation of the networked systems or an update of the learned networked system models will be shared between V&V and learning (Step 12). In case of negative evaluation of the CONNECTed system, or some problem detected during its execution, or some significant update of the learned networked system models, or some change of the networked systems or their environment, V&V provides feedback to synthesis triggering a resynthesis of the CONNECTor model and consequently

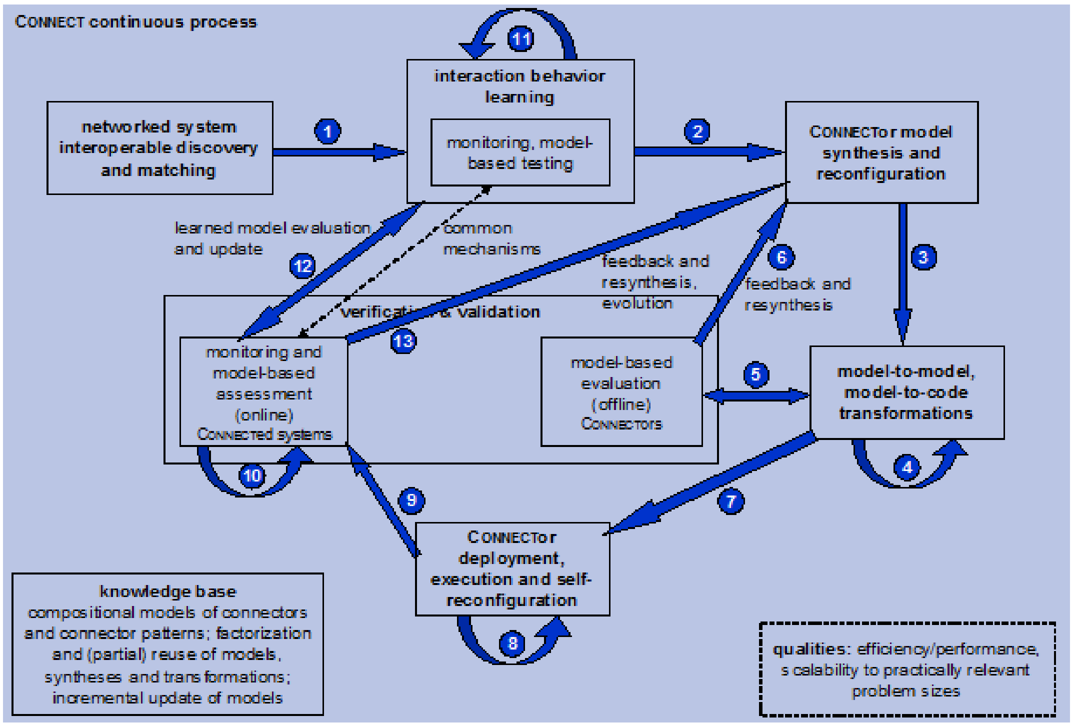


Figure 2.3: The Runtime Phases of the CONNECT Enabling Architecture

re-execution of all the steps that follow (Step 13). This puts in place a continuous CONNECT process that takes account of change and evolution towards enabling eternal systems.

These phases make up the behaviour executed by the CONNECT enablers. We describe more concretely how the enablers perform this functionality in the refinement of the CONNECT architecture.

2.3 Refinements Roadmap

Overall, three important architectural principles were identified in the original CONNECT architecture:

1. The role of *models*. Models are essential to the CONNECT process. Models are communicated, learned, synthesized, transformed, verified and reconfigured throughout the CONNECTed system lifecycle. Models are compositional and reusable. Such models constitute the shared knowledge among CONNECT actors in the CONNECT environment. Accumulating and reusing this knowledge is essential in CONNECT. A knowledge base of models and operations on models is envisioned in CONNECT, accessible and shared among CONNECT enablers.

2. *Enablers*. An extensible set of functions that provide the necessary behaviour to implement an interoperability solution. Importantly, this is an extensible set to allow richer functionality to be added in the future.

3. CONNECTors. Dynamically generated software to execute interoperability between two heterogeneous systems; importantly, these are transparent to the legacy networked systems.

The focus of this report is to make the CONNECT architecture sufficiently concrete such that initial prototype software can be developed and then evaluated to measure the effectiveness of the architectural choices. Here we specify the key refinements to the above architectural principles in order to reach this objective.

- First, and most importantly, we specify the CONNECTor architecture, i.e., how a CONNECTor is created, implemented and deployed. CONNECTors are the key requirement to enable interoperation, and hence their concrete specification is fundamental to defining the other elements of the architecture.

- The model of a network system is the common information that is utilised and exchanged between CONNECT enablers. Here we refine the Networked System Model to a set of core descriptions and present the specification languages to support this.

- The Enabler Architecture is refined to explain how enablers are deployed and communicate with one another.

It is important to identify that we do not (at this stage) refine the architecture with respect to i) non-functional requirements of the networked systems, ii) the dependability of the solution, and iii) advanced synthesis and learning behaviour. These are described further in Section 7.2 and will be included in the subsequent deliverable (D1.3) describing the next version of the CONNECT architecture, as was originally planned in the project.

2.4 CONNECTors

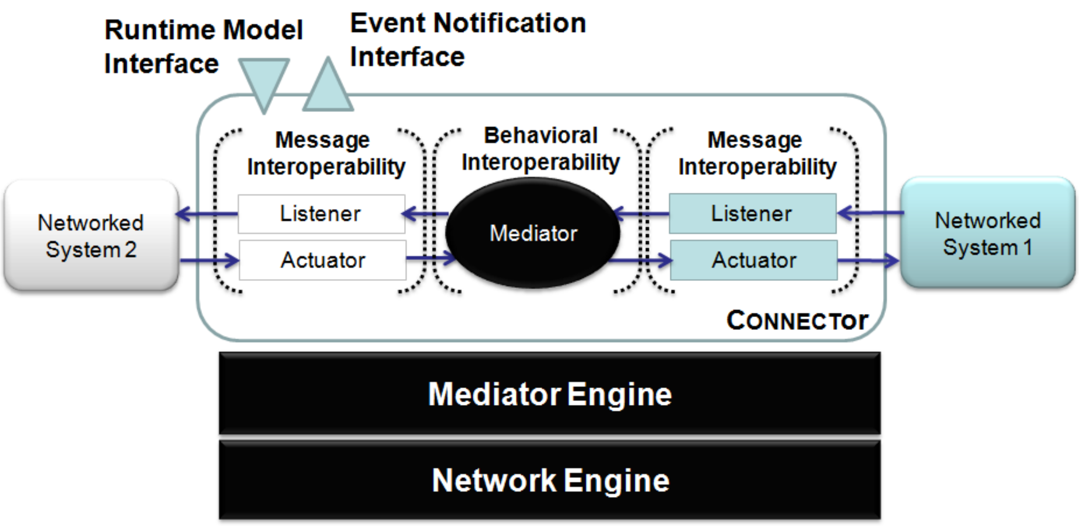


Figure 2.4: The CONNECTor Architecture

2.4.1 The Architecture of CONNECTors

We define the software elements that make up an individual CONNECTor and also how they interact in order to achieve interoperability. This CONNECTor architecture is illustrated in Figure 2.4. The software elements are described as follows:

- A *Listener* receives network messages (from the network engine) in the form of data packets and parses them according to the message format employed by the protocol that this message is specified by. Hence, each Listener parses messages from a single protocol, e.g., the SOAP listener parses SOAP messages. A listener produces an *Abstract Message* that contains the information found in the original data packet, providing a uniform representation that can be manipulated and understood by the other elements in the CONNECTor architecture. The API of the listener in Java is shown in Figure 2.5, the packet in a byte array is passed to the `MessageParse` method and a Java Object (`AbstractMessage`) representing the Abstract Message is produced.

- An *Actuator* performs the reverse role of a listener, i.e., it composes network messages according to a given middleware protocol, e.g., the SOAP Actuator creates SOAP messages. Actuators receive the Abstract Message and translate this into the data packet to be sent on the network via the network engine. The API of the actuator in Java is shown in Figure 2.5, a byte array is produced when the `AbstractMessage` object is passed to the `MessageCompose` method.

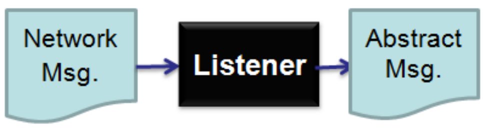
- The *Mediator* forms the central co-ordination element of a generated CONNECTor. Its role is to translate the content received from one protocol (using `Abstract Message`) into the content required to send to the corresponding protocol. The mediator therefore addresses the challenges of: different message content and formats, and different protocol behaviour, e.g., sequence of messages.

- The *Network Engine* provides a library of transport protocols with a common uniform interface to send and receive messages. Hence, it is possible to receive messages and send messages from multicast (e.g. IP multicast), broadcast and unicast transport protocols (e.g. UDP and TCP). The uniform interface provided by the network engine is illustrated in Figure 2.6.

- The *Mediation engine* is an optional element of the architecture depending upon the implementation approach taken for mediator. The behaviour of the mediator is determined by a high-level model determining the operations to take. In the case where this model is generated directly into code there is no need for a mediation engine. In the case where the mediator model is an executable model (e.g., a BPEL specification, or an alternative CONNECT mediator model) then it is the mediation engine which executes these scripts. This flexibility in the intermediary architecture allows us to investigate the benefits of the two approaches, i.e., to investigate the performance gains of direct code generation, versus the ability to easily adapt the behaviour of the CONNECTor at runtime using the mediation engine.

- Then *Event Notification Interface* outputs all important events that occur during the operation of the CONNECTor, e.g., the receiving of a network packet, the completion of parsing, etc. Other elements of the CONNECT architecture can then subscribe to receive these events; this behaviour is closely related to the monitoring and dependability enablers (as discussed in the Deliverable 5.2 [15]).

- The *Runtime Model Interface* is a Meta-Object Protocol based interface that supports introspection of the software elements that compose an individual CONNECTor, i.e., it is possible to observe at runtime what listeners, actuators and mediator are in operation. Furthermore, the interface supports the runtime adaptation of the CONNECTor architecture through the replacement of the prior described elements.




```
AbstractMessage MessageParse(byte[] dataPacket);
```


```
byte[] MessageCompose(AbstractMessage msg);
```

Figure 2.5: Listeners and Actuators API

2.4.2 Abstract Messages

A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements (typically found in message headers) and elements specific to a given message (e.g.,

```java
public interface INetworkTransport {

    public enum Protocol{TCP, UDP}

    /**
     * Create a network socket for the given parameter information.
     *
     * @param IPAddress The address of the socket
     * @param port  The port number for the socket
     * @param proto The protocol type to use {TCP or UDP}
     * @param isMcast is it a multicast socket (t) or unicast (f)
     * @return The ISocket interface to the created socket
     * @see ISocket
     */
     public ISocket newSocket(String IPAddress, int port, Protocol proto,
             boolean isMcast);

     /**
      * Receive a Packet object from the socket (blocking receive).
      * @param socket The socket to receive a msg from
      * @return The Packet data - from address and byte array
      * @see Packet
      */
     public Packet Receive(ISocket socket);

     /**
      * Send a msg to the given socket.
      * @param socket The socket to send a msg on.
      * @param msg The byte array containing the network message
      * @see Packet
      */
     public void Send(ISocket socket, byte[] msg);
}
```

Figure 2.6: The Network Engine Interface

in the message body). A CONNECTor must extract relevant elements from the received message and use them to create one or more messages according to the target protocols. Similarly, it must extract relevant elements from the received responses and ultimately create a response according to the source protocol. Hence, the design of CONNECTors is based upon these message-based events; and the key design principle is to derive information from network messages and then describe them in a protocol independent manner. We term this protocol independent description the *Abstract Message*. Received network messages are converted to an Abstract Message, correspondingly the Abstract Message is used to build the network message to be sent.

The schema for the Abstract Message content is illustrated in Figure 2.7. This shows that an Abstract Message consists of a set of fields; a field can be either primitive or structured. A *primitive field* is composed of a label naming the field, a type describing the type of the data content, a length defining the length in bits of the field, a boolean stating if this is a mandatory or optional field, and the value of the field, i.e., the data content. A *structured field* is composed of multiple primitive fields. For example, a URL field is composed of four primitive fields: the protocol, the address, the port, and the resource location.

Abstract Messages then represent the interface between the Listeners, Actuators and the Mediator, and the underlying network messages. In order to achieve interoperability dynamically, the CONNECTor receives network messages from a networked system (in the format of the protocol employed by this legacy system). This event will trigger the execution of the Mediator, whose behaviour will determine the sequence of actions that manipulate the listeners and actuators. For example, it may receive one or more messages in the Abstract Message format and it may send one or more messages by composing a new Abstract Message and sending this to an Actuator to be delivered to the target networked system.

```
<xsd:schema>
   <xsd:element name="Field">
     <xsd:complexType>
         <xsd:sequence>
              <xsd:element name="label" type="xsd:string"/>
              <xsd:element name="length" type="xsd:integer"/>
              <xsd:element name="type" type="xsd:string"/>
              <xsd:element name="mandatory" type="xsd:boolean"/>
              <xsd:element name="value" type="xsd:any"/>
              <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
   </xsd:element>

   <xsd:element name="AbstractMessage">
     <xsd:complexType>
         <xsd:sequence>
              <xsd:element name="Name" type="xsd:string"/>
              <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 2.7: The Abstract Message Schema

2.5 CONNECT Networked System Model

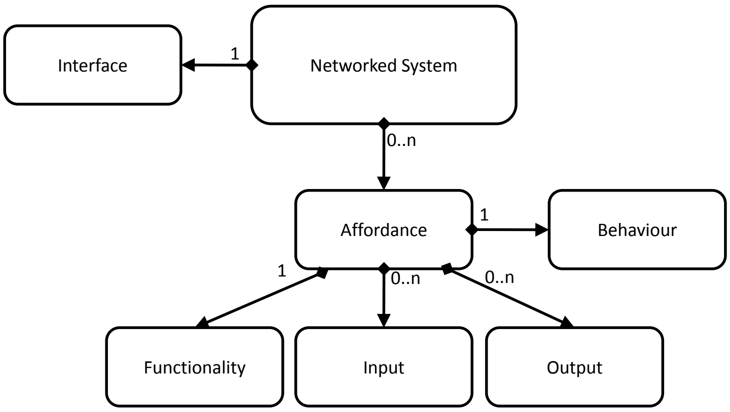


Figure 2.8: Overview of the Networked System Model

In this section we introduce and provide a short overview of the refined **Networked System Model**, emphasizing the central role it plays in the CONNECT architecture; *the full model and its complete details are provided in Section 3.2 of this document* and richly covers the description languages created for CONNECT to specify a networked system. Figure 2.8 highlights the key points of the model (with the full model illustrated in Figure 3.1):

- The *affordance* is a macroscopic view, or the quality of a feature, of a networked system. Essentially the affordance describes the high-level roles a networked system plays, e.g., ‘prints a document’, or ‘sends an e-mail’. This allows semantically equivalent action-relationships/interactions with another networked system to be matched; in short, they are doing the same thing. *A detailed definition of affordance is given in Section 3.2.2, along with the reasons for embracing it within the* CONNECT *project.*

- *Interfaces* provide a refined or a microscopic view of the system by specifying finer actions or methods that can be performed by/on the networked system, and used to implement its affordances. Each networked system is associated with a unique interface. *The **xDL language** used to specify interfaces in* CONNECT *is introduced in Section 3.2.1 of this document.*

- The *Behaviour description* documents the application behaviour in terms of how the actions of the interface are co-ordinated to achieve the system's affordance, and in particular how these are related to the underlying middleware functions. A BPEL-based specification language is employed to specify this behaviour; *further details of this description is documented in Section 3.2.2. of this document*

There have been many approaches and languages proposed for the description of services; however, these typically focus on a particular angle, e.g., the interface syntax, data formats, co-ordination, or the semantic meaning of data. To achieve interoperability there is a need for proper integration of these concepts. Hence, the role of the Networked System Model is to provide a formally-grounded, central specification that describes the syntax, behaviour and semantics of a networked system in a common description language. This common model then enables the CONNECT enablers to achieve their objectives. Importantly, we identify that ontologies are the pillar to establish a common understanding of the specification of networked systems. We see the role of ontologies as crosscutting the CONNECT architecture; rather than using ontologies to simply discover and match service descriptions, the ontologies are novelly employed down to the matching and mapping of communication protocols. *We highlight the use of ontologies in the* CONNECT *architecture in greater detail in Section 6 of this document.*

2.6 The CONNECT Enabler Architecture

The Enabler architecture is the configuration of the enabler components which are deployed in the network environment. Figure 2.9 illustrates how these combine to achieve the particular goal of CONNECT, i.e., to take two networked systems whose heterogeneity denies them from interoperating with one another, and then deploying the required CONNECTor. We discuss the individual enablers in turn, and then finally describe how they communicate in order to reach the goal.

2.6.1 The Discovery Enabler

Output to Learning Enabler: Networked System Model of one networked system.

Output to Synthesis Enabler: Matched Networked System Models of two networked systems to create a CONNECTor for.

The Discovery Enabler receives both the advertisement messages and lookup request messages that are sent within the network environment by the networked systems. The enabler obtains this input by listening on known multicast addresses (used by legacy discovery protocols). These messages are then processed; information from the legacy messages is extracted and the Networked System Models of the systems in the environment are produced. Further information about this extraction is given in Section 3.4.3. At this stage the model is composed of the affordance and the interface description. Initial matching is then performed to determine whether two networked systems are candidates to have a CONNECTor generated between. On a match, the CONNECT process is initiated; first the current model is sent to the Learning enabler, which adds the behaviour description to the model. On completion of the model, the Discovery Enabler sends the model to the Synthesis enabler.

A richer description of the Discovery Enabler is presented in Section 3 of this document.

2.6.2 The Learning Enabler

Input from Discovery Enabler: Networked System Model of one networked system (with no or partial behaviour specification).

Output to Discovery Enabler: Networked System Model of one networked system (with completed

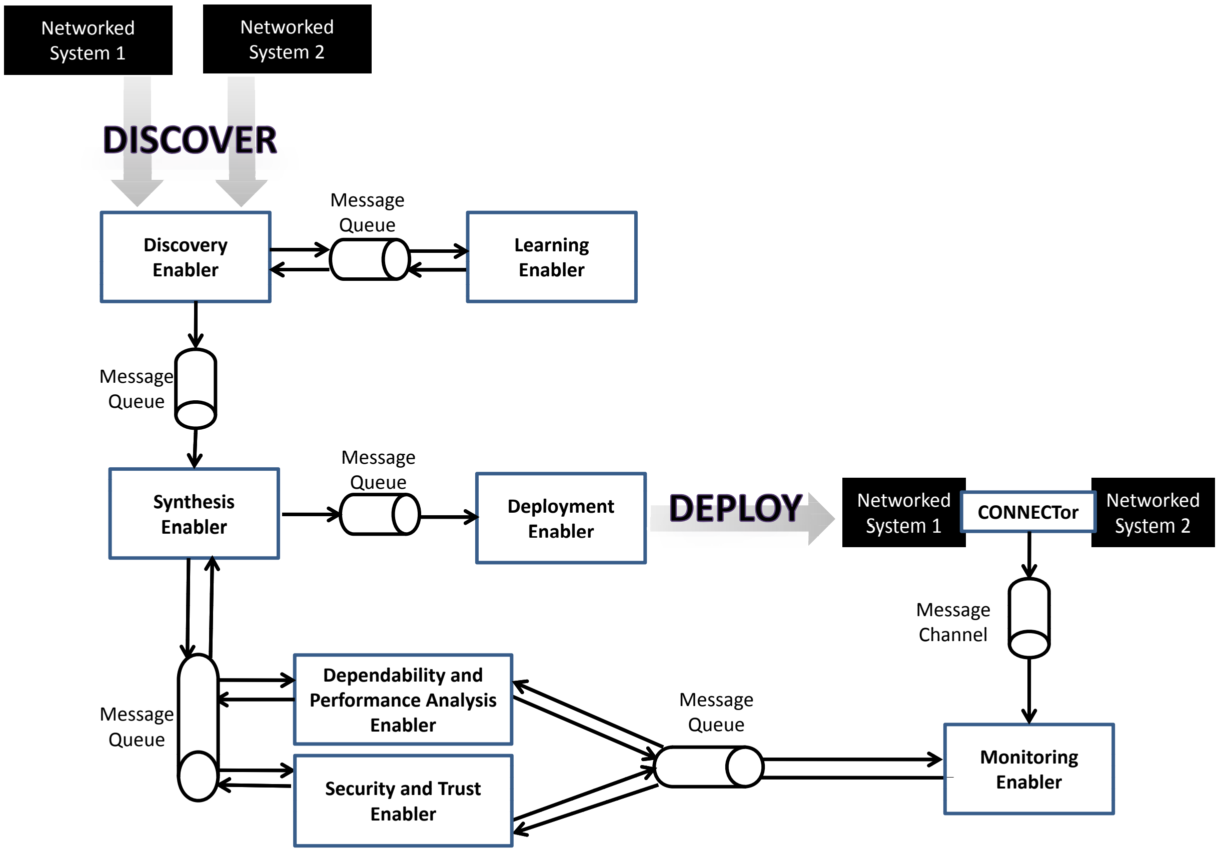


Figure 2.9: The CONNECT Enabler architecture

behaviour specification).

The Learning Enabler uses active learning algorithms to dynamically determine the interaction behaviour of a networked system from its intermediary representation and produces a model of this behaviour in the form of a labeled transition system (LTS); this employs methods based on monitoring and model-based testing of the networked systems to elicit their interaction behaviour. The implementation of the enabler is built upon the LearnLib tool [55]. It utilises the semantic annotations of the interface description from the Networked System Model as an input, and an LTS describing the interaction behaviour is produced and added to the behaviour section of the Networked System Model, and the outcome is a complete - as far as possible - instantiated networked system model. This is sent back to the Discovery Enabler to complete the discovery of the description of networked systems.

The learning enabler will build on the technologies developed in WP4 as discussed in Deliverable D4.2 [17] and folded into the architecture as results mature.

2.6.3 Synthesis Enabler

Input from Discovery Enabler: Matched Networked System Models of two networked systems to create a CONNECTor for.

Input from Dependability Enabler: Dependability assessment of a CONNECTor specification.

Output to Deployment Enabler: Constructed CONNECTor in one of the deployment formats (generated code or executable model).

Output to Dependability Enabler: CONNECTor specification (LTS model).

The role of the synthesis enabler is to take the Networked System Models of two systems and then synthesize the mediator component that is employed by the CONNECTor. For this it performs automated behavioural matching (as opposed to functional matching) and mediation of the two models [62]. This uses the annotated ontology information from the models to say where a sequence of messages is equivalent; based upon this, the matching and mapping algorithms determine an LTS model that represents the mediator. The synthesis enabler can then output two alternatives (depending upon the style of CONNECTor in use):

- Mediator code. The synthesis enabler generates the Java executable code that can be deployed directly as part of a CONNECTor configuration.
- An LTS model. The LTS model can be sent directly, in order for it to be used by the mediation engine of a CONNECTor.

Either of these two outputs is sent to the deployment enabler in order to complete the construction of the CONNECTor.

Further information about the operation of the Synthesis enabler created by WP3 is found in Deliverable D3.2 [18].

2.6.4 Deployment Enabler

Input from Synthesis Enabler: Constructed CONNECTor in one of the deployment formats (generated code or executable model).

The Deployment Enabler receives as input the mediator code and the original Networked System Models; its objective is to finalise and then deploy the CONNECTor in this case. In order to do this, the enabler executes two important roles:

- It composes the required functionality to ensure that CONNECTors will communicate with the legacy networked systems, i.e., it will add the listeners and actuators to the mediator generated by the Synthesis Enabler. We discuss how the listeners and actuators are realised in Section 4 of this Deliverable.
- It deploys and manages the executable code of the CONNECTors in the network. For this, the enabler utilises OSGi techniques; *more information about the deployment solution is provided in Section 5 of this document.*

2.6.5 Dependability and Performance Analysis Enabler

Input from Synthesis Enabler: CONNECTor specification (LTS model) and Networked System Models (non-functional properties description).

Output to Synthesis Enabler: Dependability assessment of a CONNECTor specification.

Input from Monitoring Enabler: Data derived from real-time execution of CONNECTors.

Once a CONNECTor specification has been produced by the synthesis enabler it sends it to the dependability and performance analysis enabler to determine if the non-functional requirements (as described in the Networked System Model of each networked system) are satisfied. If so, the enabler tells the synthesis enabler to go ahead and deploy; otherwise, the dependability enabler creates a set of enhancements to the specification and returns these to the synthesis enabler. The dependability enabler also continuously determines if the CONNECTor maintains its non-functional requirements. It receives monitoring data from the monitoring enabler and in the case where there is no longer compliance, the dependability enabler sends a new specification to the synthesis enabler to initiate redeployment of a suitable CONNECTor in the current conditions.

A full description of the behaviour of the dependability and performance analysis enabler created by WP5 is given in Deliverable D5.2 [15].

2.6.6 Security and Trust (SXT) Enabler

Input from Synthesis Enabler: CONNECTor specification (LTS model) and Networked System Models (non-functional properties description).

Output to Synthesis Enabler: Security and Trust CONNECTor specification.

Input from Monitoring Enabler: Data and messages derived from real-time execution of CONNECTors.

Once a CONNECTor specification has been produced by the synthesis enabler it sends it to the security and trust enabler (SXT) to determine if the non-functional requirements (as described in the Networked System Model of each networked system) are specified. If so, the enabler creates a set of enhancements to the specification and returns these to the synthesis enabler. The SXT enabler also continuously determines if the CONNECTor maintains its non-functional requirements. It receives monitoring data from the monitoring enabler and in the case where there is no longer compliance, the enabler sends a new specification to the synthesis enabler to initiate redeployment of a suitable CONNECTor in the current conditions.

A description of trust and security requirements are given in Deliverable D5.2 [15], whereas a full description of the SXT enabler will be addressed in future work.

2.6.7 Monitoring Enabler

Input from CONNECTors: Raw data concerning state information of CONNECTors.

Input from Dependability Enabler: Request to monitor a deployed CONNECTor.

Output to Dependability and Performance Analysis Enabler: Data derived from the real-time execution of CONNECTors.

The monitoring enabler receives requests concerning which CONNECTors to monitor and then collects raw information about the CONNECTors by monitoring data that this CONNECTor published to the monitoring channel. The derived data is passed to the dependability enabler to determine if the original non-functional requirements are being matched.

A full description of the behaviour of the Monitoring enabler created by WP5 is given in Deliverable D5.2 [15].

2.6.8 The CONNECT Message Bus

The enablers and CONNECTors use a simple message-based communication model to exchange information with one another. In this intermediary architecture, we use the Java Messaging Service (JMS) implementation from Oracle[1] to implement the Message Bus. The reason for the choice of communication model is that two styles of communication are important to CONNECT and are both provided by the technology:

- Point-to-Point exchange between enablers. As described earlier, the enablers send content (e.g., models and code) to be processed by a specific party, e.g., the discovery and learning enabler communicating to build the Networked System Model. JMS allows the behaviour to be achieved using a message queue as illustrated in Figure 2.9.

[1] http://www.oracle.com/technetwork/java/index-jsp-142945.html

- Publish-Subscribe communication of CONNECTor behaviour. The CONNECTors produce events in order for them to be monitored; enablers can subscribe to the channels that the CONNECTors publish these events to. For example, in Figure 2.9 the monitoring enabler subscribes to this channel in order to monitor CONNECTor events.

2.7 Conclusion

To summarise, this section has identified the significant progress that has been made in the second year towards realising a more concrete architecture. The specification of the CONNECT Networked System Model offers the central model around which the CONNECT architecture is described. There is also a clear picture of how the enablers collaborate in order to realise the actual CONNECTors. Yet there remains the important issue of non-functional properties; initial work on considering interoperability solutions that takes into account non-functional requirements has begun, cf., the dependability, security and trust, and monitoring enablers in the enabler architecture, and also the monitoring features of the deployed CONNECTors themselves. However, this remains clearly work in progress and further integration of the work carried out in WP5 is a key objective in the work to be carried out in the third year.

3 The Discovery Enabler

3.1 Introduction

In order to achieve interoperability between independently-developed systems, it is imperative, before any further communication can occur, that pairs of systems, which are potentially compatible—by virtue of their complementary functionality—can be mutually discovered. The CONNECT *discovery enabler* is specifically responsible for enabling *compatible* networked systems to discover each other, assuming networked systems make use of some *resource discovery protocol* (a.k.a. *service discovery protocol*) to advertise their presence in the network and also receive the advertisements of peers.

In its most basic form, a *Service Discovery Protocol* (*SDP*) decomposes into:

- The *Directory node* that is an optional, logically centralized node that caches service advertisements;
- The *Client nodes* that seek provided networked services by issuing dedicated requests in the network;
- The *Provider nodes* that advertise provided services in the networks they join.

Two other core constituents of an SDP are the language used for describing service advertisements and requests, and the *matchmaking* that sets the conditions under which an advertisement is *compatible* with (or *matches*) a request. Then, the design of an SDP may vary in the following dimensions [39]:

- *Service description language and associated matchmaking*, which respectively define the service meta-model for the description of service requests and advertisements, and associated matching relations. The service description may range from a simple list of (attribute, value) pairs to detailed definitions specifying the service's interface signature, behavior and quality of service.
- *System structure*, which may be centralized (single directory node), decentralized (no directory node, leading to peer-to-peer discovery) or hierarchical (structured, distributed directory nodes).
- *Discovery method*, which may be push-based (i.e., the provider nodes pro-actively advertise their presence towards directory or client nodes), pull-based (i.e., provider nodes react upon requests by client nodes), or symmetric (both client and provider nodes pro-actively contribute to the discovery process).

According to the above, the literature is rich with SDPs whose design is customized according to the target networking environment. For instance, we refer the interested reader to: UDDI for Web services [53], Jini for intranets [1], UPnP/SSDP for home networks [34], Ariadne for MANET [58], EASY for pervasive semantic Web services [47], and energy-efficient SDP for multi-radio networks [11]. Furthermore, several projects have investigated interoperability solutions for SDPs so that nodes in common networks or in bridged networks, but belonging to distinct discovery domains (i.e., using heterogeneous SDPs), are able to locate each other. These include in particular solutions from the consortium partners such as REMMOC [28] from Lancaster University, and INDISS [8] and MUSDAC [57] from INRIA. However, while existing SDPs allow addressing a large variety of networking environments, they target rather homogeneous networked systems from the standpoint of their interaction paradigms. Indeed, to the best of our knowledge, most of the existing SDPs are concerned with client-service based systems where interaction is message-based. Furthermore, matchmaking is often associated with simple descriptions of networked systems, i.e., a list of (attribute, value) pairs for the simplest case [31] or interface signatures when the SDP is coupled with an overall middleware solution for distributed computing [1]. More complex service descriptions have been considered as part of open pervasive networking environments, such as Web services and pervasive computing environments. Solutions then include semantic service description using Semantic Web technologies [43, 32] as well as behavioral specification using concurrent models [29]. However, while matchmaking of semantic service descriptions has been extensively studied [37], few discovery protocols integrate semantic service matchmaking [48]. Similarly, while behavioral matchmaking of networked services has deserved significant attention over the last years [29], its integration within dynamic discovery protocols remains the exception [52]. Last but not least, few protocols consider the combined exploitation of semantic and behavioral knowledge about networked systems [9].

Nevertheless, in the CONNECT context, the discovery enabler requires a detailed description of networked systems so as to enable their CONNECTion despite heterogeneity in the protocols used for interaction, from the application down to the middleware layers. Indeed, as detailed in Deliverable D3.2 [18], CONNECTor synthesis relies on the description of networked systems in terms of interface signatures together with affordances and the associated behavior. Even if this requirement can hardly be fulfilled by current SDPs, thanks to the various CONNECT enablers networked system descriptions may be partly learned, hence enabling the CONNECTion of legacy systems, provided that they advertise their presence in the network by way of a legacy SDP.

This section introduces the design of the CONNECT discovery enabler together with its first prototype implementation, which enables *matching/compatible* networked systems to be located, where matching/-compatibility is defined with respect to the mediation patterns supported by the CONNECTor synthesis process [18]. The proposed enabler builds upon the extensive literature in the area of service discovery protocols, including their interoperability. However, the CONNECT discovery enabler distinguishes itself by (i) dealing with the discovery of highly heterogeneous networked systems, and further by (ii) integrating advanced semantic and behavioral matchmaking according to the matching relations introduced in Deliverable D3.2 [18]. Specifically, the CONNECT discovery enabler:

- Defines an *ontology-based language* for the abstract, yet precise, description of networked systems' observable behavior, by building upon methodologies of the Semantic Web Services domain [37] and of concurrency theory to reason about the interoperability of networked systems. Compared to state-of-the-art networked system descriptions, the proposed language in particular allows the definition of interface signatures and behavioral (process) descriptions of highly heterogeneous networked systems by handling different coordination models corresponding to different types of middleware. Indeed, existing interface description languages target client-service based systems and hence lack the expressiveness required by today's networking environment. This is for instance witnessed by the ad hoc extensions of WSDL to deal with other interaction paradigms (e.g., publish-subscribe extension of WSDL by REMMOC in [28]), beyond traditional RPC-like communication.

- Features *ontology-based behavioral matchmaking* regarding the functional and non-functional properties of networked systems, as well as their respective interaction behaviors. However, the current version of the discovery enabler focuses on functional and behavioral matchmaking, while non-functional matchmaking is area for future work, which will build upon WP5 results.

- Provides an extensible solution for *universal discovery* using protocol plugins to manage legacy discovery protocols, in a way similar to state-of-the-art middleware solutions for discovery protocol interoperability [48, 56]. Plugins further include a custom plugin for CONNECT-aware nodes so as to enable networked system advertisements using our ontology-based description language.

- Integrates with the other CONNECT enablers so as to support the CONNECTion of *compatible/matching* networked systems. Precisely, integration with the *learning enabler* allows the learning of the interaction behaviors of networked systems, as they are in general not readily available from the interface descriptions used by legacy discovery protocols. Integration with the *synthesis enabler* then enables the actual CONNECTion of CONNECTable systems through the synthesis of appropriate mediators, while the *dependability enablers* enforce matching with respect to non-functional properties.

The next section introduces the CONNECT Networked System Description Language, simply called *NSDL*, for the ontology-based description of networked systems' observable behavior. Associated matchmaking is presented in Section 3.3, which builds upon the matching relations introduced in Deliverable D3.2 for the sake of CONNECTor synthesis [18]. The architectural design of the CONNECT discovery enabler follows in Section 3.4, in particular: (i) dealing with the discovery of both CONNECT-aware networked systems (i.e., systems that use NSDL to advertise their presence in the network) and networked systems using legacy SDPs to join networks, and (ii) discussing integration with other CONNECT enablers such as the learning and synthesis enablers. The first prototype implementation of the discovery enabler is then presented in Section 3.5. Finally, conclusions are drawn in Section 3.6, summarizing the status of the discovery enabler development and ongoing and future work towards completing integration within the CONNECT architecture.

3.2 CONNECT Networked System Description Language

As stated in Deliverable D1.1 [6], tremendous research effort is taking place on ensuring interoperability in open, pervasive networking environments. Building upon results of the Semantic Web Service domain, we posit that ontologies are the fundamental basis for establishing a common understanding of the specification of networked systems, while the behavioral specification of networked systems is central to analysing interoperability, including possible mediation. Then, in a manner similar to ontologies for Web services, i.e., OWL-S [43] and WSMO [32], a networked system is described in terms of the following elements (see Figure 3.1):

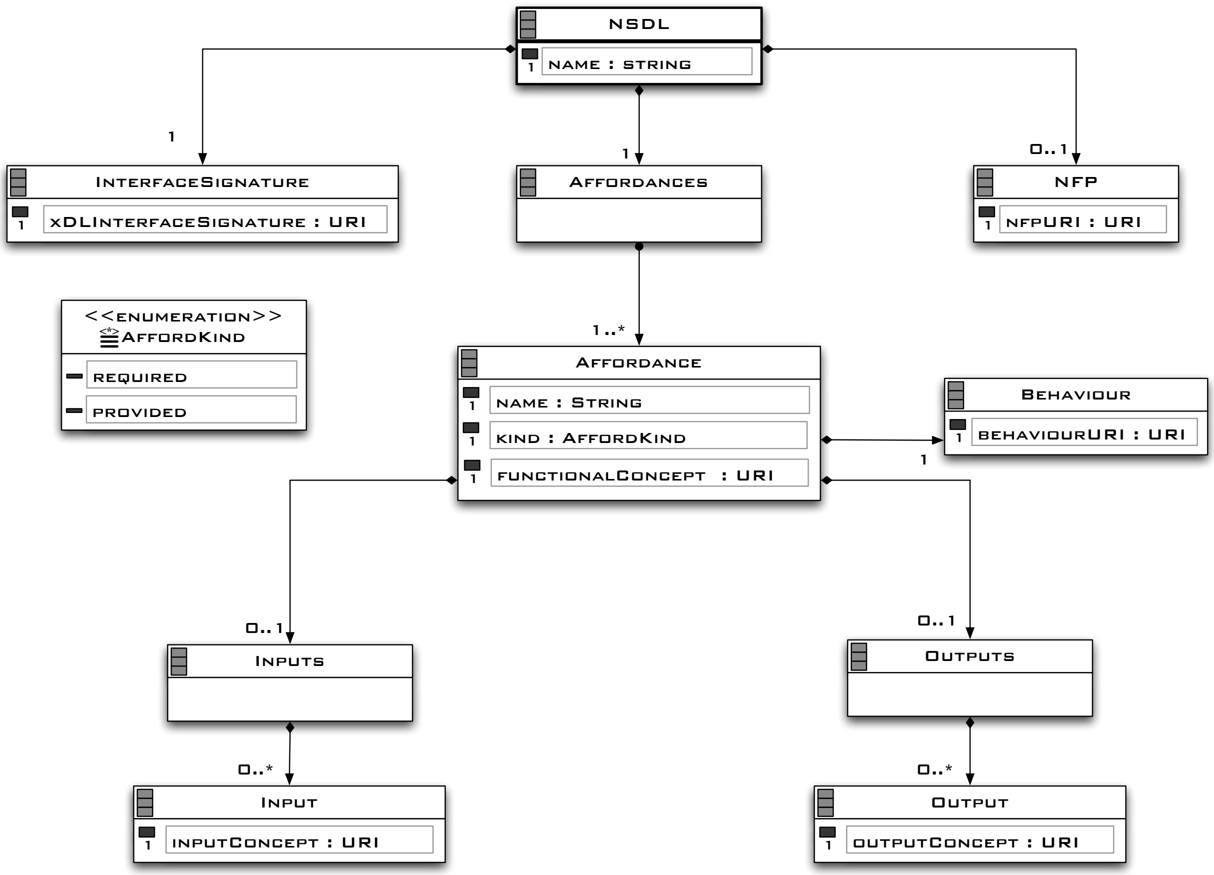


Figure 3.1: Networked system description

- **Interface signature** (simply called interface): There is one interface associated with each networked system. The interface defines the observable actions of the networked system. Then, as prompted by the Future Internet vision [20] that includes the Internet of Content, Things, Services and even People, we need to address CONNECTion among highly heterogeneous networked systems. In particular, networked systems rely on different coordination paradigms depending on their target domains (e.g., sensors networks rely on data-centric communication while information systems are more likely to be client/service-based). Such diversity is part of the observable behavior of networked systems although this is in general hidden in classical service/interface description language due to close coupling with the underlying middleware and its interaction paradigms. As a result, the description of networked systems must make the interaction paradigms assumed for any interaction explicit. The definition of interface signatures is further detailed in Section 3.2.1.

- **Affordance**: The affordances specify the *high-level functionality* of the networked system and are implemented as protocols over the system's observable actions. The notion of *affordance* corresponds to that of *capability* from the Semantic Web service domain. However, we use another term to stress that we deal with interoperability beyond the Web service domain, although acknowledging that we build upon the state of the art of the domain. The definition of affordances is further detailed in Section 3.2.2.

- **Non-functional properties (NFP)**: Expressing the non-functional properties of networked systems is one of the CONNECT challenges that is addressed in WP5. To the best of our knowledge, the literature lacks a generic framework in which properties such as dependability, performance, security and trust have been considered together. Therefore, a Property MetaModel has been defined (see Deliverable D5.2 [15]) to express all these properties. In short, the meta-model decomposes in five parts: (i) Property Specification that mainly provides the name and the type (i.e., Quantitative or Qualitative) of a given property; (ii) Metric Specification that defines the method to assess a property; (iii) Metric Template that expresses the way to compute metrics; (iv) Event Specification that describes, if needed, specific time events when the properties have to be monitored; and finally (v) Metrics Domain for which a property is relevant or needs to be verified or guaranteed.

For illustration, in the following, we consider the photo sharing scenario that is also used in Deliverable D3.2 [18]. Briefly stated, the photo sharing scenario concentrates on ad hoc photo sharing within a public space and more specifically on the ad hoc CONNECTion between two mobile versions of the photo sharing application. Both versions allow upload, download and commenting on photo files. However, the two versions differ as follows: (i) one implements peer-to-peer photo sharing using tuple-space-based shared memory and (ii) the other implements centralized photo sharing through interaction with a dedicated server and distinguishes between clients that upload and those that download photos so that the former operation is performed only by authorized users.

3.2.1 Interface Signature and Binding Definition

As discussed previously, the networked system's interface defines the observable actions performed by the system for interaction with its environment and hence for implementing its affordances. Compared to existing interface definition languages, the proposed XML-based schema for interface signature definition, called *xDL* (extensible Description Language), tackles different coordination models so as to account for the diversity of today's networking environment.

In detail, the definition of xDL is inspired by WSDL and its extension SAWSDL[1]; it is thus composed of three parts, all including semantic annotations (see Figure 3.2):

- **Types**: Serve to define the interface's data types (i.e., set of *Type* elements) in a way similar to SAWSDL to which the interested reader is referred for detail.

- **Primitives**: Enable the characterization of the observable actions of the networked system. An action (or primitive) is defined by both the corresponding application-specific and communication actions, the latter relating to the communication model implemented by the underlying middleware. The description of primitives is further detailed in the following Sections 3.2.1 to 3.2.1.

- **Bindings**: Define the data format and protocol details supporting the defined primitives. Precisely, a binding named *name*, gives: the URI where the specific encoding of data is defined through the *data* item, the URI of the protocol used for executing the primitive through the *primitive* item, and the *address* of the networked system as a URI. There may be a number of bindings for a given primitive, depending on the specific middleware used.

[1] http://www.w3.org/TR/sawsdl/

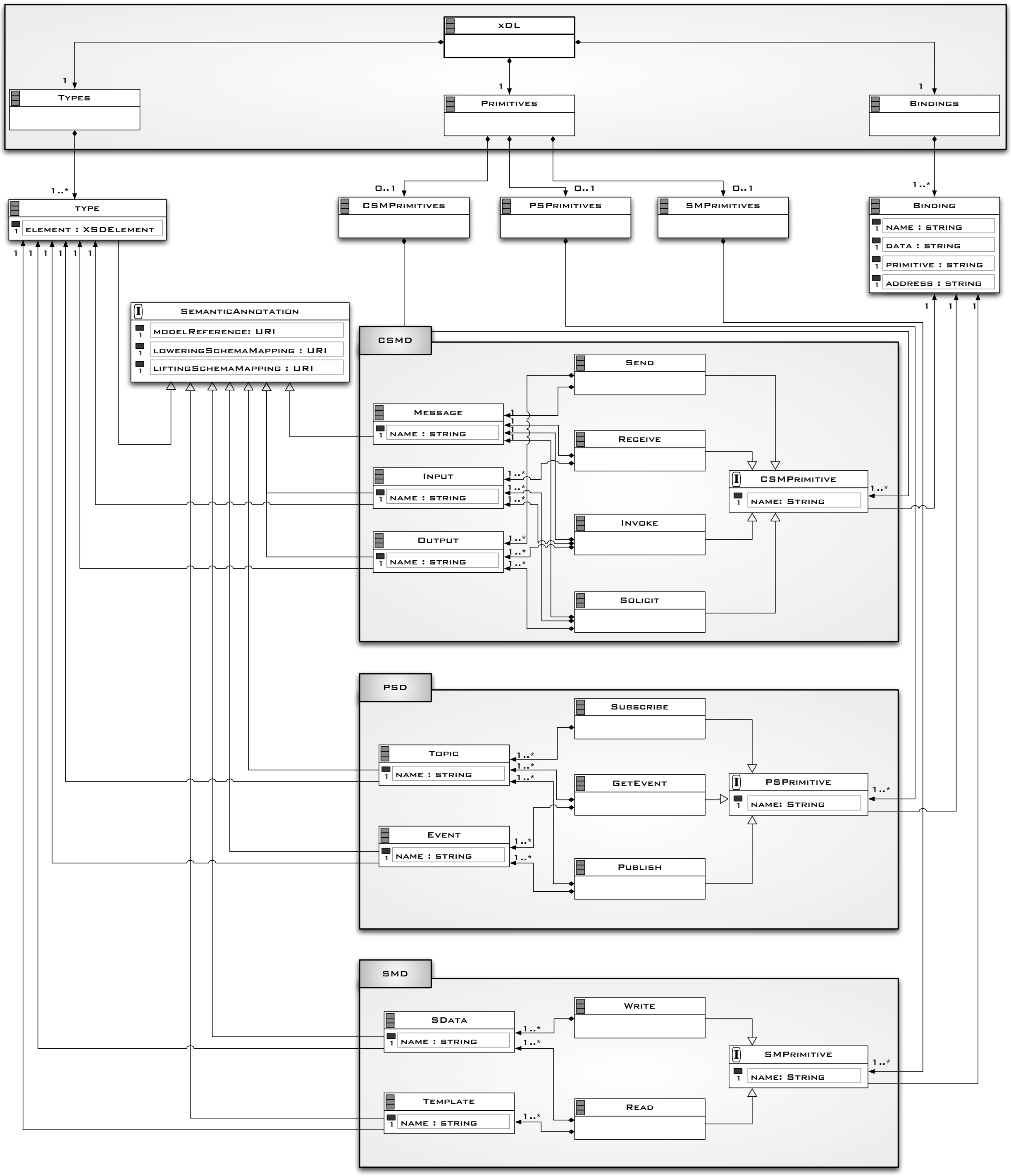


Figure 3.2: Networked system interface signature and binding

Regarding *semantic annotations*, we use *modelReference* to specify the association between an xDL attribute and a concept in a given ontology, as well as *liftingSchemaMapping* and *loweringSchemaMapping* to define the transformation from (to) the XML Schema of the attribute to (from respectively) its associated semantic description. Concretely, we rely on a description-logic-based ontology language, namely OWL[2] (Web Ontology Language), to describe formally the concepts used by the interacting parties.

As discussed above, the primitives executed by networked systems depend on the application-specific actions that are implemented together with related middleware-specific communication actions. In general, middleware-specific communication actions are hidden from the SDP's interface definition language because either actual interaction with the service is considered to be beyond the scope of the SDP (e.g., SLP [31]) or the SDP is closely tied to a middleware platform and hence middleware-specific communication actions may be automatically inferred (e.g., Jini [1]). However, in today's networking environment where networked systems are highly heterogeneous and may need to be composed on the fly, the interface definition of networked systems must make explicit the communication paradigms associated with application-specific actions. The definition of the communication paradigms then derives from that of reference coordination models. According to the classification of connectors given in [64] and the related reference middleware ontology introduced in Deliverable D3.2 [18], we identify the following reference coordination models for the definition of xDL: *client-service*, *message-orientation*, *publish-subscribe* and *shared memory*. This leads us to introduce dedicated descriptions as part of xDL (see Figure 3.2), where we merge client-service and message-oriented communication paradigms, as defined below.

CSMD: Client/Service and Message-oriented Description

The CSMD description firstly allows the definition of one-way message-based communications:

- *Send (message, output)*: denotes the sending of a *message* with *output* as content.
- *Receive (message, input)*: denotes the receipt of a *message* with *input* as content.

In the above definition, the application-specific semantics of the action (or primitive) is given by the ontology concepts (i.e., semantic annotations) associated with the message and its content, while the semantics of the communication action is given by that of *send*/*receive*. Note that in the above, a message may denote an operation in the context of the client-service coordination model, in which case the operation is called asynchronously and the output of the send function corresponds to the operation's input parameters.

Further, with regard to the client-service communication paradigm, the CSMD description additionally defines:

- *Invoke (message, input, output)*: denotes the synchronous invocation of an operation, characterized by *message*, with *input* and *output* parameters;
- *Solicit (message, input, output)*: denotes the matching counterpart of *Invoke* on the service provider side;

whose application-specific semantics also derives from the concepts associated with message, inputs and outputs while the communication semantics is that of *invoke*/*solicit*.

PSD: Publish/Subscribe Description

Under the the publish/subscribe model, networked systems communicate asynchronously [49, 10]: systems acting as publishers produce events on certain topics; other systems acting as subscribers receive the published events that match the topics to which they subscribe. Then, the application-specific semantics of actions under the publish/subscribe model is defined by that of the topics and events, while the semantics of communication relies on the following middleware functions:

- *Subscribe (topic)*: denotes the subscription to a *topic* so as to receive asynchronous events that match the *topic* expression.

[2] http://www.w3.org/TR/owl2-overview/

- *getEvent (topic, event)*: denotes the consumption of a received asynchronous *event* that matches the given *topic*.
- *Publish (topic, event)*: denotes the publication of an *event* matching a given *topic*.

SMD: Shared Memory Description

Using the shared memory coordination paradigm, networked systems communicate by reading and writing data into some shared data space, which may be centralized or distributed. Then, the application-specific semantics of actions under the shared memory model is given by the data being manipulated, while the semantics of the communication is given by the operation performed on the shared data space, i.e.:

- *write (data)*: denotes the writing of *data* into the shared data space.
- *read (template, data)*: denotes the reading of *data* matching *template* from the shared data space.

Example

As an illustration, Appendix 8.1 gives the xDL description of the interfaces of the two photo sharing applications under consideration, which are respectively based on the client-service and shared memory interaction paradigms, while Figures 3.3 and 3.4 give their graphical representation. In the figures, we note the semantically annotated descriptions of the communication actions, which specify the application-specific semantic of actions performed via the given primitives. The centralized photo sharing implementation in Figure 3.3 is specified using CSMD. Therefore, the client-side application actions are invoked though the *Invoke* function, while they are processed on the server side using the *Solicit* function. The producer first authenticates by invoking the *Authenticate* operation then calls the *UploadPhoto* operation in order to upload a photo. The consumer searches for photos based on criteria over their metadata using the *SearchPhoto* operation, then he can download a photo (*DownloadPhoto*) or a comment about this photo (*DownloadComment*). The consumer can also comment a photo through the *CommentPhoto* operation. Finally, the actions of the Photo Sharing server are complementary to the client actions. The binding is performed using SOAP. The peer-to-peer-based implementation in Figure 3.4 is specified using SMD and defines a single interface signature, as all the peers feature the same observable actions. The peers producing a photo, *Write* the corresponding metadata and file tuples in the tuple space. Then other peers, can search for photos matching some criteria by performing a *Read* over the tuple space. They can then select one photo and *Read* the corresponding file or comment. Note that the *PhotoMetadata*, *PhotoFile*, and *PhotoComment* associated with the same *Photo* have the same *photoID*. The binding is based on LIME.

3.2.2 Affordance Definition

While networked systems interact through the observable actions defined in their interface signatures, they actually CONNECT according to high-level functionality that they provide and require within the network. We call such functionalities *affordance*s, while acknowledging the similar meaning of the notion of *capability* from the Semantic Web service domain. As depicted in Figure 3.1, page 35, and according to the networked system model introduced in Deliverable D3.2 [18] and recalled in Section 2, an *affordance* is specified in terms of:

1. The *ontology-based semantic* characterization of the high level functionality implemented by the affordance, i.e., the ontology concepts associated with its functionality (*functionConcept*), and inputs (set of *inputConcept*) and outputs (set of *outputConcept*) parameters. In addition, the *kind* item serves defining whether the affordance is *required* or *provided* in the network.
2. The affordance's *behavior*, that is, the process or protocol executed by the networked system to coordinate with its environment to realize the given high-level functionality. In a first step, we assume that the coordination takes place with a single peer while the implementation of affordance through composition of multiple networked systems is an area for future work. Likewise, we do not consider affordance nesting.

CSMD Server Description		
Solicit		
operation	Authenticate	
input	login	string
output	authenticationToken	string
binding	SOAPSolicitAuthenticate	
Solicit		
operation	UploadPhoto	
input	photo	Photo
	authenticationToken	string
output	acknowledgment	string
binding	SOAPSolicitUploadPhoto	
Solicit		
operation	SearchPhoto	
input	photoMetadata	PhotoMetada
output	photoMetadataList	PhotoMetadaList
binding	SOAPSolicitSearchPhoto	
Solicit		
operation	DownloadPhoto	
input	photoID	string
output	photoFile	PhotoFile
binding	SOAPSolicitDownloadPhoto	
Solicit		
operation	DownloadComment	
input	photoID	string
output	photoComment	PhotoComment
binding	SOAPSolicitDownloadComment	
Solicit		
operation	CommentPhoto	
input	photoID	string
output	photoComment	PhotoComment
binding	SOAPSolicitCommentPhoto	

CSMD Producer Description		
Invoke		
operation	Authenticate	
input	login	PhotoMetada
output	photoMetadata	PhotoMetada
binding	SOAPInvokeAuthenticate	
Invoke		
operation	UploadPhoto	
input	photoMetadata	PhotoMetada
	authenticationToken	string
output	photoMetadata	PhotoMetada
binding	SOAPInvokeUploadPhoto	

CSMD Consumer Description		
Invoke		
operation	SearchPhoto	
input	photoMetadata	PhotoMetada
output	photoMetadataList	PhotoMetadaList
binding	SOAPInvokeSearchPhoto	
Invoke		
operation	DownloadPhoto	
input	photoID	string
output	photoFile	PhotoFile
binding	SOAPInvokeDownloadPhoto	
Invoke		
operation	DownloadComment	
input	photoID	string
output	photoComment	PhotoComment
binding	SOAPInvokeDownloadComment	
Invoke		
operation	CommentPhoto	
input	photoID	string
output	photoComment	PhotoComment
binding	SOAPInvokeCommentPhoto	

Figure 3.3: CSMD-based photo sharing interface

According to the definition of the overall CONNECT architecture, the specification of the affordance behavior may be automatically learned by the CONNECT learning enabler. However, CONNECT-aware networked systems may alternatively publish the specification of their behavior so as to ease their CONNECTion. This leads us to introduce a supporting language for the user-friendly specification of the affordance's behavior, for which we build upon BPEL[3]. Indeed, BPEL allows the specification of concurrent processes while being supported by tools for user-friendly editing as well as translation into formal concurrent models (e.g., WS-Engineer[4]). However, BPEL, being tightly coupled with Web services, supports limited communication primitives and remote interface description (i.e., WSDL). Nevertheless, the base language can be extended with new activities, which are called extension activities (EAs). We thus exploit this feature to introduce the elements of the different coordination models supported by xDL. Furthermore, by building upon the flexibility of XML[5], these extensions can be generic and be refined later in unlimited ways according to concrete middleware platforms. Figure 3.5 gives the definition of the BPEL EAs associated with xDL, which are direct from the xDL definition of Figure 3.2.

As an illustration, Appendix 8.2 gives the BPEL specification of the behavior of the two photo sharing affordances under consideration, which respectively interact using the client-service (CSMD-based) and shared memory (SMD-based) models, while Figure 3.6 gives their graphical representations. In the SMD-based description, there is a single affordance to realize photo sharing: each networked system acts as a peer that may read and write photo files as well as annotate shared photo files with comments. On the other hand, the CSMD-based description introduces three affordances, which relate to the centralized photo sharing service (or server) at the bottom of the figure and to the photo sharing clients that may either download and comment (top right of the CSMD description), or upload (top left of the CSMD description) photos.

[3] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[4] http://www.doc.ic.ac.uk/ltsa/eclipse/wsengineer/
[5] http://www.w3.org/TR/xmlschema11-1/

SMD Description		
write		
data	photoMetadata	PhotoMetadata
binding	LimeOutPhotoMetadata	
write		
data	photoFile	PhotoFile
binding	LimeOutPhotoFile	
read		
template	photoIDTemp	PhotoMetadata
data	photoComment	PhotoComment
binding	LimeRdPhotoComment	
read		
template	photoIDTemp	PhotoMetadata
data	photoComment	PhotoComment
binding	LimeInPhotoComment	
write		
data	photoComment	PhotoComment
binding	LimeOutPhotoComment	
read		
template	photoDetailsTemp	PhotoMetadata
data	photoMetadataList	PhotoMetadata
binding	LimeRdgPhotoMetadata	
read		
template	photoIDTemp	PhotoMetadata
data	photoFile	PhotoFile
binding	LimeRdPhotoFile	

Figure 3.4: SMD-based photo sharing interface

3.3 CONNECT Matchmaking

Given the NSDL description of networked systems, their matchmaking follows from the specification of the matching relations introduced in Deliverable D3.2 [18]. Briefly stated, the matchmaking of networked systems towards their CONNECTion is first determined according to the ontology-based semantic matching of their affordances. Then, if two networked systems implement semantically matching affordances, the enabler checks whether the actions they perform for the realization of the affordances match semantically, by mapping their interfaces according to base mediation patterns. If so, it is ultimately verified whether the protocols/processes defining the behaviors of the affordances may indeed coordinate given the computed interface mapping. Then, in the case of successful behavioral matchmaking, the two networked systems may be CONNECTed and the associated CONNECTor shall implement a *mediator* that performs the elicited interface mapping. As part of WP3, we are currently investigating efficient algorithms for on-the-fly computation of interface mapping and associated behavioral matchmaking.

Within WP1, as a first step to the implementation of the CONNECT enablers within the CONNECT architectural framework, we have implemented behavioral matchmaking assuming 1-to-1 semantic mapping of actions, i.e., any application-specific action executed as part of the realization of an affordance maps to a single application-specific action of the peer's matching affordance.

The following section recalls the ontology-based semantic matchmaking of affordances introduced in Deliverable D3.2 [18]. Then, Section 3.3.2 details behavioral matchmaking under 1-to-1 ontology-based semantic mapping of application-specific actions and taking as input the behavioral specification of affordances using the BPEL specification introduced in the previous section.

3.3.1 Ontology-based Semantic Matchmaking of Affordances

Matching on the basis of the ontology-based semantics of affordances permits the efficient selection of potentially *compatible* systems from the wide range of networked systems known to the discovery enabler. Systems that have semantically matching affordances then have their interface and behaviors analyzed in

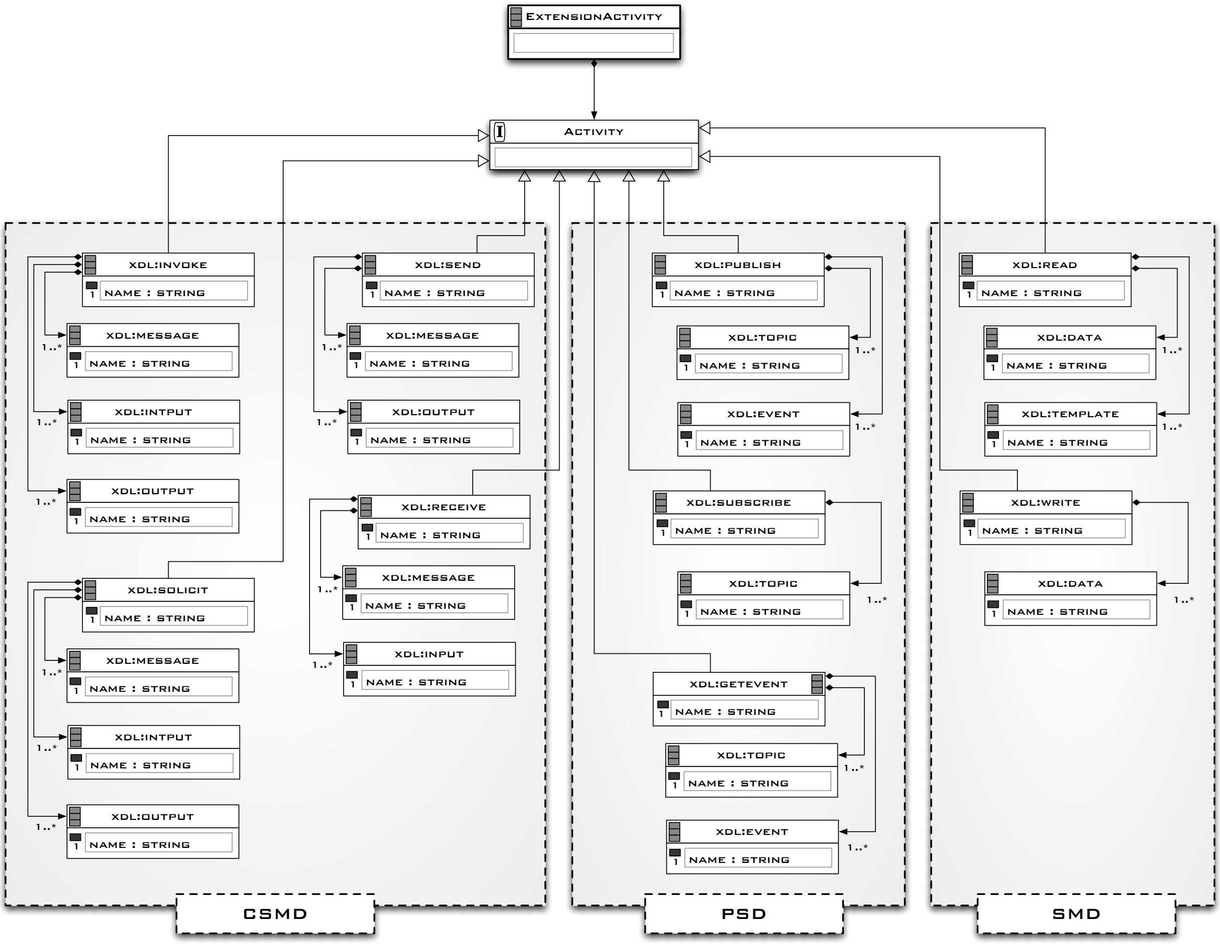


Figure 3.5: Extension activities associated with xDL communication paradigms

detail to determine if they do indeed match at this lower level.

Matching of a pair of affordances is performed according to the rules defined in D3.2 [18]. Given a pair of affordances:

$$Aff_1 =< Required, f_1, I_1, O_1 > \text{ and } Aff_2 =< Provided, f_2, I_2, O_2 >$$

where $f_{i \in 1,2}$ refers to the function concepts, $I_{i \in 1,2}$ are the input concepts, and $O_{i \in 1,2}$ are the output concepts);

Aff_1 *semantically matches* Aff_2 iff:
f_1 is subsumed by f_2, I_2 is subsumed by I_1 and O_1 is subsumed by O_2, according to the given ontology,

where a concept C *is subsumed by* a concept D in a given ontology O, written $C \leq_{onto} D$, if in every model of O the set denoted by C is a subset of the set denoted by D.

Only pairs of semantically matching affordances are worth considering further for behavioral matching and possible CONNECTOR synthesis as defined in D3.2 [18], although weaker matching could be considered in the case when degenerate behavior is acceptable in the given application context.

3.3.2 Behavioral Matchmaking of Affordances

Given semantically matching affordances, their associated protocols must be checked for the potential to coordinate, possibly under some mediation implemented by the supporting CONNECTOR. Advanced

Figure 3.6: Behavior of the photo sharing affordances

matching relations and supporting algorithms are being investigated within WP3 so as to facilitate CONNECTion between heterogeneous systems while not sacrificing dependability. In particular, WP3 investigates CONNECTor synthesis and related matching relations in a way that supports most mediation patterns (i.e., merge/split, missing/extra and reordering of actions) [18]. However, in order to enable early experimentation, we have been implementing behavioral matchmaking considering basic mediation patterns with respect to application-specific actions, as presented below.

Supported Mediation Patterns

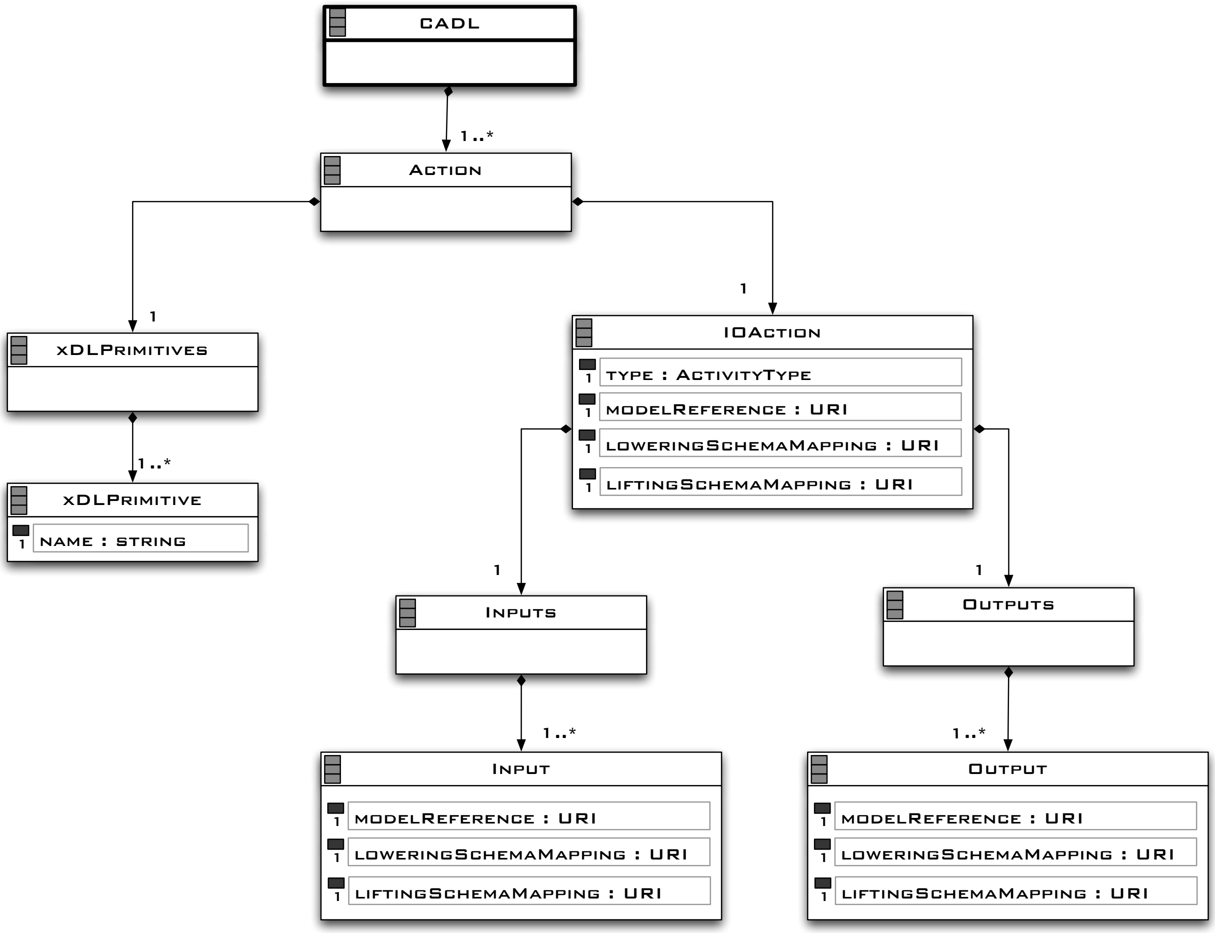


Figure 3.7: CONNECT Action Definition Language

The behavioral matchmaking implemented in the current CONNECT discovery enabler accounts for the following mediation patterns:

- *Mapping between middleware-specific communication actions* of heterogeneous coordination models. According to Deliverable D3.2 [18], such mapping relies on making the networked systems' actions middleware-agnostic, that is, middleware-specific communication actions are mapped onto abstracted input and output communication actions. The resulting actions are characterized using the XML-based CADL (CONNECT Action Definition Language) language given in Figure 3.7. Then, following the mapping introduced in Deliverable D3.2 [18], Figure 3.8 defines the mapping of xDL primitives to CADL input and output actions. The *Solicit*, *Send*, *Write*, and *Publish* functions are mapped onto output actions while the *Invoke*, *ReceiveMessage*, and *Read* functions are translates into input actions. The sequence of a *Subscribe* function followed by a *GetEvent* function is mapped to an input action. The rationale behind this mapping is that the output actions represent the production of an application action whereas the input actions are meant for it consumption. CADL hence offers a uniform view of the observable actions since it specifies for each action from the interface the xDL primitive, the middleware-agnostic action, *IOAction* which can either be of type

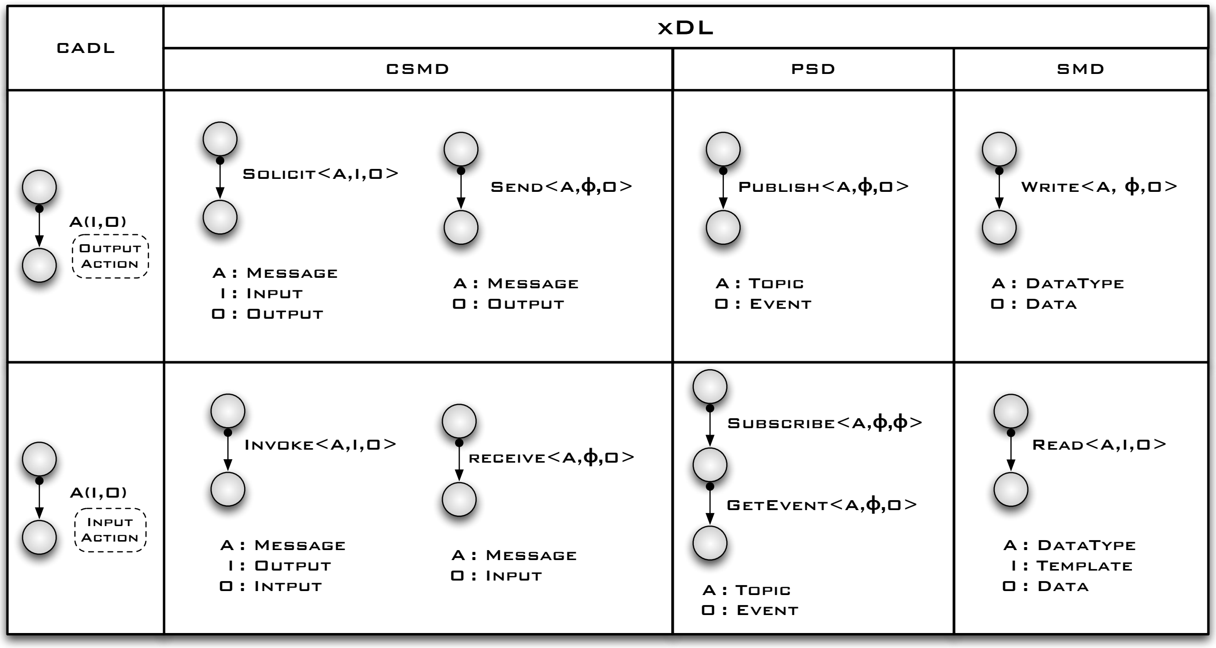


Figure 3.8: From xDL actions to middleware-agnostic CADL actions

input or *output* and the associated *Inputs* and *Outputs* parameters. Each constituent of the action is associated with a *ModelReference* referring to its semantic concept, *LoweringShemaMapping* and *LiftingSchemaMapping* that specify the transformations between the syntactic (XML-based) types and the associated semantic concepts.

- *Mapping between application-specific actions* according to their ontology-based semantics, where we consider only one-to-one mapping: such mapping derives from the *is-a* relation (a.k.a. subsumption) defined over concepts in the corresponding ontology.

Following the above, any protocol defining an affordance behavior may be translated into a middleware-agnostic protocol where communication actions are defined in terms of CADL actions. In addition, application-specific actions are matched against each other according to their semantics and *subsumption* relationships holding in the related ontology. Note that given the definition of CADL in Figure 3.7, the original xDL definition is referred to in the description of the middleware-agnostic action using the *xDL primitives* item so as to keep track of the translation; this information will subsequently be used by the mediator synthesis process as implemented by the synthesis enabler.

Ontology-based Model Checking for Behavioral Matchmaking

Given the above, an appealing analysis technique to reason about behavioral matching of networked systems is model checking. Considering two networked systems, NS_1 and NS_2, and assuming that NS_1 *requires* the semantically matching affordance provided by NS_2, verifying that NS_1 behaviorally matches with NS_2 amounts to setting the traces of NS_1 as legitimate traces that must be achieved by NS_2.

Among relevant formal models, Foster [25] provides a method and associated tool to translate BPEL into the FSP (Finite State Processes) process algebra whose semantics is given in terms of labeled transition systems (LTS) [41]. However, this translation abstracts away input/output data, which are of great relevance to the behavioral matchmaking process since they give the semantics of actions. Therefore, we enhance the obtained FSP, which we call *OFSP* (Ontological FSP), with details from the CADL description of actions. However, to avoid the URIs included in the CADL description interfering with the FSP syntax, CADL actions are encoded in base 32 (see Figure 3.9). Hence, BPEL protocols defining the behavior of affordances in terms of xDL actions are translated into OFSP processes whose actions are semantically annotated using CADL actions. Although another process algebra would have worked equally well, we

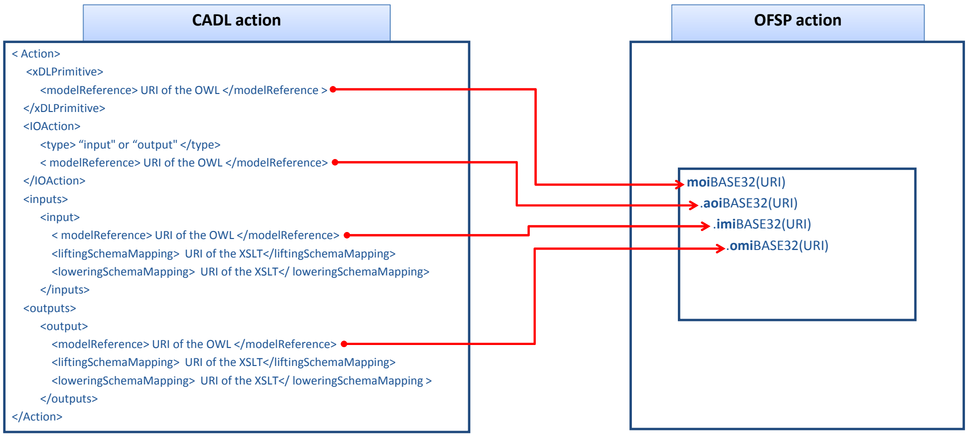


Figure 3.9: Encoding OFSP

chose FSP for its simplicity, emphasis on compositional analysis, and existing LTSA[6] (Labeled Transition System Analyzer) tool support.

Given the OFSP specification of affordance behaviors, behavioral matchmaking may be reasoned upon using safety analysis since a safety property specifies acceptable behavior, in terms of legitimate traces, for the process it is composed with. A system S will *satisfy* a property P if S can only generate sequences of actions (traces) which, when restricted to the alphabet of P, are acceptable to P [12]. LTSA performs safety analysis by computing the safety property, which is also an FSP process, composing the property with the process to be checked; then, if there is a trace from the initial state to an error state, the system is unsafe. However, LTSA compares the actions of processes against syntactic equality only, whereas we need them to be compared semantically using the CADL annotations embedded in the OFSP specification. Therefore, we have modified the safety analysis in order to introduce the subsumption relation between actions, as presented below.

Let:

- αP be the ontological alphabet of a process P, i.e., the set of all the actions P can execute, which are specified using CADL,
- P/a be the function describing the behavior of P after it has engaged in an action $a \in \alpha P$, and
- $*$ be the non-process representing an isolated node.
- a trace $s = \langle a_1, a_2, ..., a_n \rangle$ denotes the sequence of actions $a_i \in \alpha P$ in which process P engages and represents the communications with its environment,
- P/s represent the behavior of P after engaging in the whole sequence s of actions, one after the other.
- the set of all traces of P be defined as [33]:

$$traces(P) \stackrel{\text{def}}{=} \{s \mid P/s \neq *\}$$

Then, a process Q *satisfies* a property P if every trace of Q is also a trace of P:

$$P \sqsubseteq Q \stackrel{\text{def}}{=} traces(Q) \subseteq traces(P)$$

Note that this definition is similar to the one of *refinement* [33]. Indeed, a convenient way to check refinement in LTSA is by using safety analysis. However, while the original definition of trace inclusion relies on the syntactic equality of embedded actions, we extend the definition considering ontology-based semantic actions. Specifically, we evaluate the *semantic compatibility* of actions using the ontology-based notion of *subsumption*, noted $\leq_{onto}$. We recall:

[6]http://www.doc.ic.ac.uk/ltsa/

$$a_1 =< required, op_1, input_1, output_1 > \leq_{onto} a_2 =< provided, op_2, input_2, output_2 > \textit{ iff}$$
$$(op_1 \leq_{onto} op_2) \wedge (input_2 \leq_{onto} input_1) \wedge (output_1 \leq_{onto} output_2)$$

Accordingly, two traces $s_1 = \langle a_1, a_2, ..., a_n \rangle$ and $s_2 = \langle b_1, b_2, ..., b_n \rangle$ *semantically match iff* each required action of s_1 (s_2) is subsumed by a provided action of s_2 (respectively s_1) sequentially, hence the *semantic matching* of two traces is formalized as:

$$s_1 \hookrightarrow_{onto} s_2 \overset{\text{def}}{=} a_i =< required, op_i, input_i, output_i > \leq_{onto} b_i =< provided, op'_i, input'_i, output'_i >$$
$$\wedge b_i =< required, op'_i, input'_i, output'_i > \leq_{onto} a_i =< provided, op_i, input_i, output_i > \quad (1 \leq i \leq n)$$

Furthermore, using the semantic matching relation between traces, a process Q *ontologically satisfies* a property P ($P \sqsubseteq_{onto} Q$) if each trace of Q semantically match a trace of P, i.e., formally:

$$P \sqsubseteq_{onto} Q \overset{\text{def}}{=} \forall s_Q \in traces(Q), \exists s_P \in traces(P) \text{ such that } s_Q \hookrightarrow_{onto} s_P$$

Subsequently, the fact that a process Q does not satisfy a property P (noted $\not\sqsubseteq_{onto}$) is defined as follows:

$$P \not\sqsubseteq_{onto} Q \overset{\text{def}}{=} \exists s_Q \in traces(Q) \text{ such that } \forall s_P \in traces(P)\ s_Q \not\hookrightarrow_{onto} s_P$$

The sequence s_Q is, in fact, a counterexample that can be used to adapt the behavior of Q to P's behavior. This leads us to introduce the following *behavioral matching* relations between networked systems, NS_1 and NS_2, which provide or require semantically matching affordances:

- **Exact matching:** assesses compatibility for symmetric interactions such as peer-to-peer communication where both networked systems provide and require the same affordance.

$$NS_1 == NS_2 \Leftrightarrow$$
$$(NS_1 \sqsubseteq_{onto} NS_2) \wedge (NS_2 \sqsubseteq_{onto} NS_1) \text{ under mediation}$$

- **Plugin matching:** evaluates compatibility for asymmetric interactions such as client-server communication where NS_1 is providing an affordance and NS_2 requiring it.

$$NS_1 >> NS_2 \Leftrightarrow$$
$$(NS_1 \sqsubseteq_{onto} NS_2) \wedge (NS_2 \not\sqsubseteq_{onto} NS_1) \text{ under mediation}$$

- **Mismatching:** identifies behavioral mismatch as:

$$NS_1 \text{ !! } NS_2 \Leftrightarrow$$
$$(NS_1 \not\sqsubseteq_{onto} NS_2) \wedge (NS_2 \not\sqsubseteq_{onto} NS_1)$$
$$\text{under mediation.}$$

OLTSA for Automated Ontology-based Model Checking

In order to support automated ontology-based model checking, we have extended the LTSA tool with an OWL-based reasoner to achieve semantic behavioral matchmaking; we call the resulting tool *OLTSA* (Ontological LTSA).

LTSA is a free Java-based verification tool that automatically composes, analyzes, graphically animates FSP processes and checks safety and liveness properties against them. There are several verification tools such as CADP[7] that provide similar functionality as LTSA. We adopted LTSA since it is a free Java-based tool, it emphasizes composition, and provides a plugin (LTSA-WS [25]) to translate BPEL specifications to FSP processes automatically.

Figure 3.10 depicts the class diagram of the OLTSA extension to LTSA, which includes:

- **ManipulatingOntologies**: is a utility class to manipulate ontologies using OWL API[8]; it allows ontologies to be loaded either using a URL or locally by specifying a path to the ontology. It manages a `Vector` of loaded ontologies to avoid loading the same ontology many times. Then, the parameter `forceUpdate` makes it possible to update the reference to an ontology through the `getClasses` method. It offers the possibility of pre-loading all the classes of an ontology. It also permits a reasoner to be associated with a loaded ontology. We use HermiT[9] but since we are accessing the

[7] http://www.inrialpes.fr/vasy/cadp/
[8] http://owlapi.sourceforge.net/
[9] http://hermit-reasoner.com/

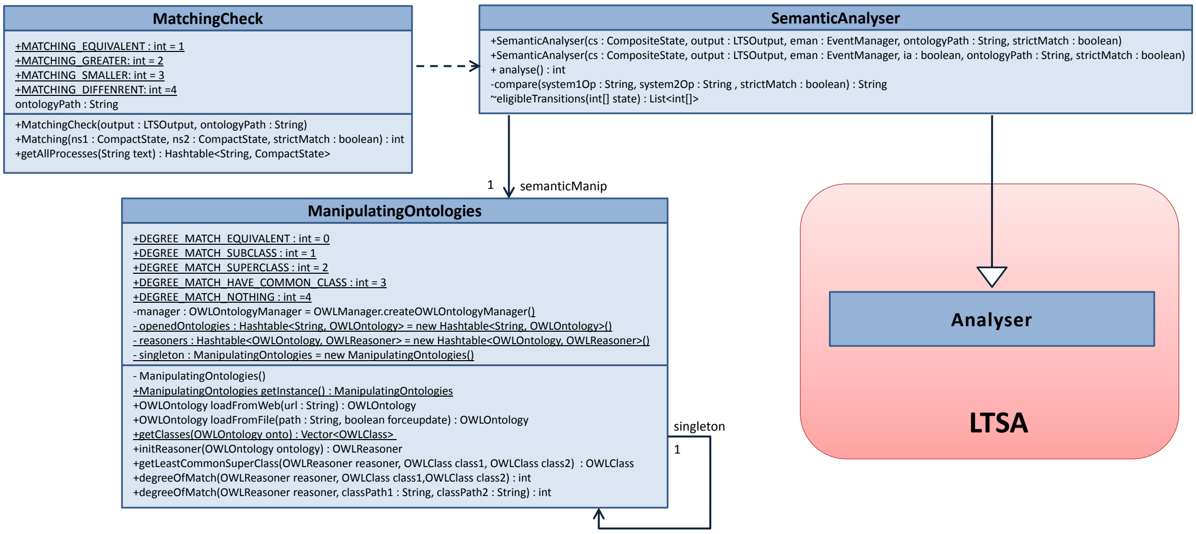


Figure 3.10: OLTSA class diagram

reasoner through the OWL API, any reasoner implementing the `OWLReasonerFactory` interface (such as FaCT++, HermiT, Pellet and Racer) would have worked equally well. Finally, it compares two OWL concepts `class1` and `class2` either using the references to these classes, if they are pre-loaded, or directly using their respective paths. The result can be:

- `DEGREE_MATCH_EQUIVALENT` if they are equivalent,
- `DEGREE_MATCH_SUBCLASS` if `class1` is subsumed by `class2`,
- `DEGREE_MATCH_SUPERCLASS` if `class1` subsumes `class2`,
- `DEGREE_MATCH_HAVE_COMMON_CLASS` if there exists a class other than `owl:Thing` that subsumes both of them, in which case the method `getLeastCommonSuperClass` allows to have this class, or
- `DEGREE_MATCH_NOTHING` if there is no semantic relationship between the two.

- **SemanticAnalyser**: performs a *semantic* safety analysis on the composition of a process and a property represented in one `CompositeState` passed to the constructor. It extends the `Analyser` class of LTSA responsible for syntactic safety analysis by implementing a semantic comparison of the labels constructed as $<$*required/provided, operation, input, output*$>$.

- **MatchingCheck**: uses the `SemanticAnalyser` in both directions to return the behavioral matching between the two networked systems, each of which specified by a process, which is a `CompactState`.

In addition to the above, the GUI of LTSA has been modified in order to include the OLTSA support for semantic behavioral matchmaking.

Note that the proposed semantic behavioral matchmaking may possibly be extended to cope with the reordering mediation pattern. Indeed, this may be achieved by making data-independent actions concurrent, either by using data-flow optimization of the BPEL process describing the behavior of the affordance [66], or by relying on automated partial order reduction [35] implemented in most model checkers, including LTSA. However, such an extension is an area for future work.

3.4 CONNECT Discovery Enabler

The design of the universal CONNECT discovery enabler builds upon the analysis of existing interoperability platforms for service discovery protocols, and in particular solutions developed by the consortium partners and experience learned from them. We have more specifically designed the CONNECT discovery enabler upon the MUSDAC [56] discovery platform, which was further extended with semantic service capabilities in order to enhance the supported matchmaking [47]. Basically, in MUSDAC[10], the environment is viewed as a dynamic composition of independent networks in which services use different protocols for discovering and accessing services. Then, MUSDAC relies on specific plugins to interact with legacy service discovery protocols, manages the efficient dissemination of the service information between the different networks, and enables clients to locate all the networked services in them.

However, the MUSDAC platform is dedicated to service-oriented systems and does not deal with behavioral matchmaking. We have thus revisited this base solution in an extensive way to support the discovery of heterogeneous networked systems and their matchmaking, using NSDL and associated matchmaking introduced in the previous sections. Last but not least, the discovery enabler integrates with the other CONNECT enablers according to the CONNECT architectural design discussed in Section 2.

3.4.1 Architecture

Building upon the MUSDAC [56] design, the architecture of the CONNECT Discovery Enabler decomposes into (see Figure 3.11):

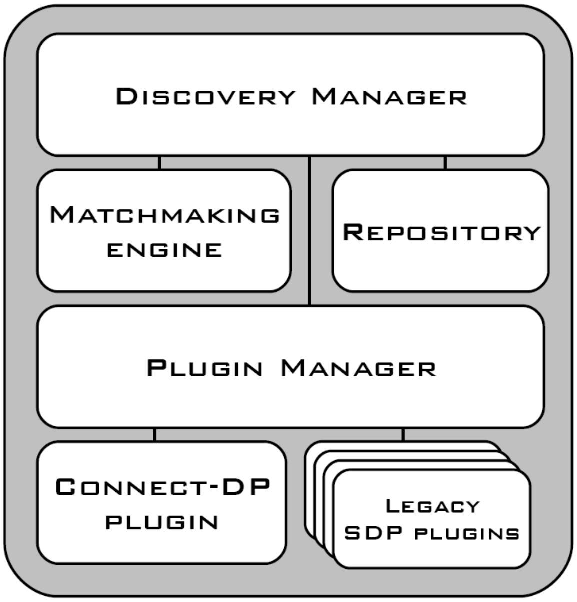


Figure 3.11: Discovery enabler architecture

- **Discovery manager:** Coordinates the various parts of the Discovery Enabler, interacting with the plugin manager and repository to store discovered networked system descriptions and initiate matchmaking.
- **Matchmaking engine:** Implements matchmaking according to the definition given in Section 3.3 and is thus no longer discussed in this section.
- **CONNECT-DP plugin:** Implements a customized service discovery protocol, called CONNECT *Discovery Protocol* (CDP for short). As detailed in Section 3.4.2, CDP may be used by CONNECT-aware nodes to advertise the networked systems they host using NSDL descriptions or query for other networked systems in the network.
- **Legacy SDP plugins**: Implements bridging with legacy SDPs so as to enable the discovery of networked systems that use legacy SDPs to advertise their presence and/or query for networked

[10] https://www-roc.inria.fr/arles/index.php/software/81-musdac-a-middleware-for-multi-protocol-service-discovery-and-access.html

Message	Parameters
Hello	endpoint
Bye	endpoint
Register	NDSL
Match	list of (affordance, endpoint)
Connect	endpoint
Connected	endpoint

Figure 3.12: CDP messages.

systems in the network. As presented in Section 3.4.3, Plugins extract legacy networked systems descriptions (e.g., WSDL, SAWSDL, UPnP, etc.) that they translate into NSDL, with the assistance of CONNECT enablers and in particular learning. As discussed in Section 3.5 on prototype implementation, our current prototype bridges with UPnP SSDP and DPWS WSDD.

- **Plugin manager:** Manages the plugin lifecycle by loading, starting and stopping the set of plugins.
- **Repository:** Caches networked system descriptions using the NSDL format together with the related semantic-aware OFSP specification of affordances behavior; the design of the repository is discussed in Section 3.4.4.

3.4.2 CONNECT Discovery Protocol

By virtue of having been designed with CONNECTor synthesis in mind, the CONNECT Discovery Protocol (CDP) supports the extraction of all the information needed by the CONNECT enablers, by exploiting NSDL. The CDP is inspired by WS-Discovery[11] (a.k.a. WSDD – Web Service Dynamic Discovery). The CDP protocol is optimized to reduce the number and the size of the exchanged packets during the discovery phase.

The main goal of CDP is to enable networked systems that are CONNECT-aware to be discovered and to provide enriched semantic and behavioral description natively, i.e., the NSDL description that defines interface, affordances and associated behaviors, and non-functional properties (Section 3.2), although dealing with non-functional properties is yet to be integrated in our design. CDP functions are currently accessible as a Web service and may easily be extended to be accessed through alternate protocols.

CDP defines two entities: the CDP server (i.e., the discovery service) that manages the CDP Clients (networked systems) that are in its network. The protocol further allows the deployment of an arbitrary number of servers. Using CDP, the CDP server/repository and CDP clients announce their presence on the network to discover one another; then, CDP clients register both required and provided affordances with the server. Then, CDP can identify provided affordances that match required ones.

Figure 3.12 lists the messages used by CDP, while Figure 3.13 illustrates the process of CONNECTion between two CONNECT-aware nodes that use CDP. First, when the Networked Systems (NS) join the network, they multicast/broadcast a *Hello* message (*Step 1*) to advertise their presence and to look for a Discovery Enabler (i.e., CDP server). When a CDP server receives a *Hello* message from a CDP Client, it answers with a unicast *Hello* message to the CDP client (*Step 2*). Then, the CDP client knows the address of the CDP server and can hence register and send its NSDL description using *Register*, which leads to storing the advertisement in the discovery enabler's repository (*step 3*). After performing a matchmaking process (i.e., seeking affordances matching the required one in the repository according to the matchmaking definition of Section 3.3), if the Discovery Enabler finds at least a matching networked system, a *Match* message is sent to the requester node (i.e., the one requiring an affordance) with the list of end points of networked systems that provide matching affordances (*step 4*). The CDP client sends a *Connect* message (step 5) containing the end point of the selected networked system. This leads the discovery enabler to interact with the synthesis, dependability and deployment enablers so as to generate the needed CONNECTor. Then, once the required CONNECTor is deployed, the CDP server informs the networked systems with the *Connected* message (*step 7*).

[11] http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html

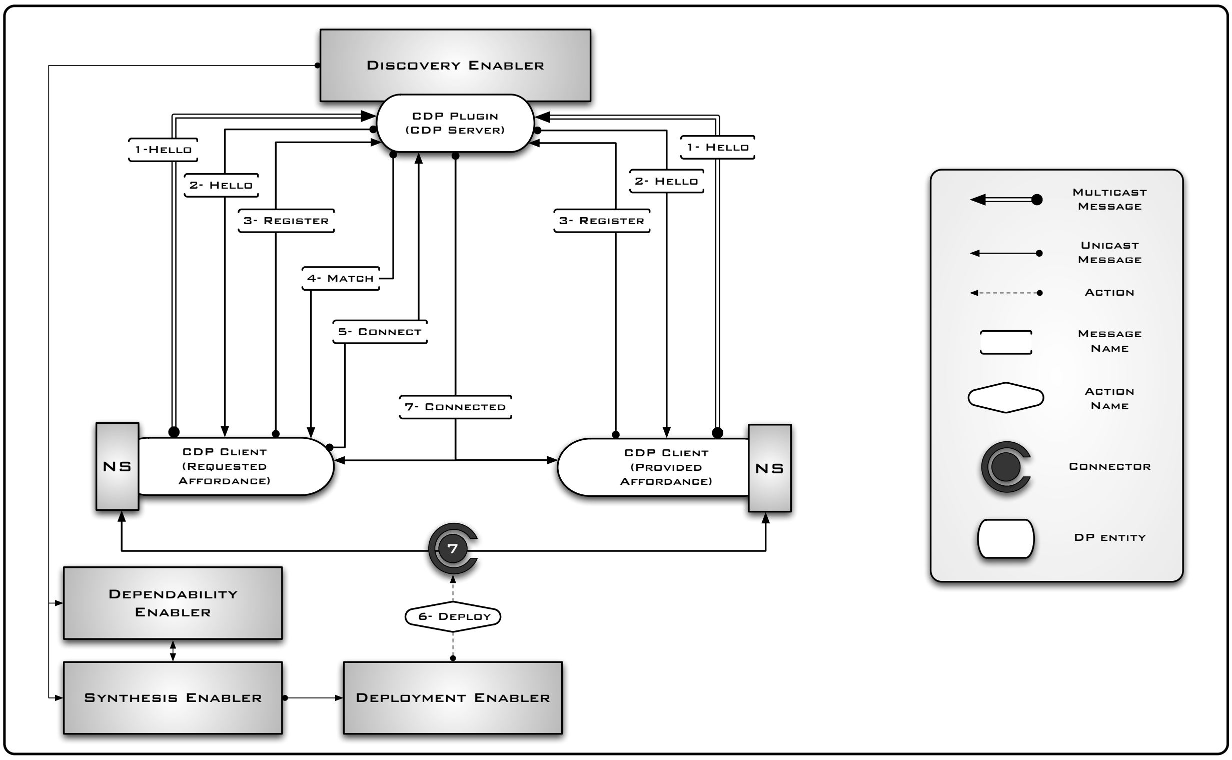


Figure 3.13: The CONNECT discovery protocol

3.4.3 Legacy Plugins

The SDP plugins interface with legacy discovery protocols deployed in the network, and further translate networked system descriptions from the specific legacy format to NSDL descriptions (see Section 3.2). However, networked systems that implement legacy protocols often provide only their syntactic interface signature (e.g., WSDL) and do not provide any means to extract their behavior nor their non-functional properties. Neither do they provide ontology-based descriptions, although this may be eased by latest standards like SAWSDL (Semantic Annotations for WSDL[12]). This then requires automated learning about the networked systems' behavior and properties, as supported by the CONNECT learning enabler developed in WP4.

	CONNECT-aware NS Description	Legacy NS Description
Affordance	Provided	Inferred
Syntactic Interface	Provided	Provided
Semantic Interface	Provided	Inferred
Behavior	Provided	Learned (WP4)
Non-functional properties	Provided	Learned (WP4)

Figure 3.14: NS description extraction

Figure 3.14 summarizes how networked system description is extracted considering both CONNECT-aware Networked Systems (NS) and legacy ones. In the former case, systems provide NSDL descriptions by employing CDP while in the latter, a simple interface description is provided via the legacy discovery protocol. Focusing on legacy service descriptions, ontology-based semantic characterization of affordances and interfaces may be inferred from the name and documentation of primitives, through process-

[12] http://www.w3.org/TR/sawsdl/

ing with dictionaries and ontologies [26, 38, 7]. Such a feature is not yet integrated in our prototype and is part of our ongoing work through collaboration within the context of the EternalS project[13]. In addition, active learning, as investigated within WP4 [17], can be used to learn the behavior as well as the non-functional properties of legacy networked systems given their interface.

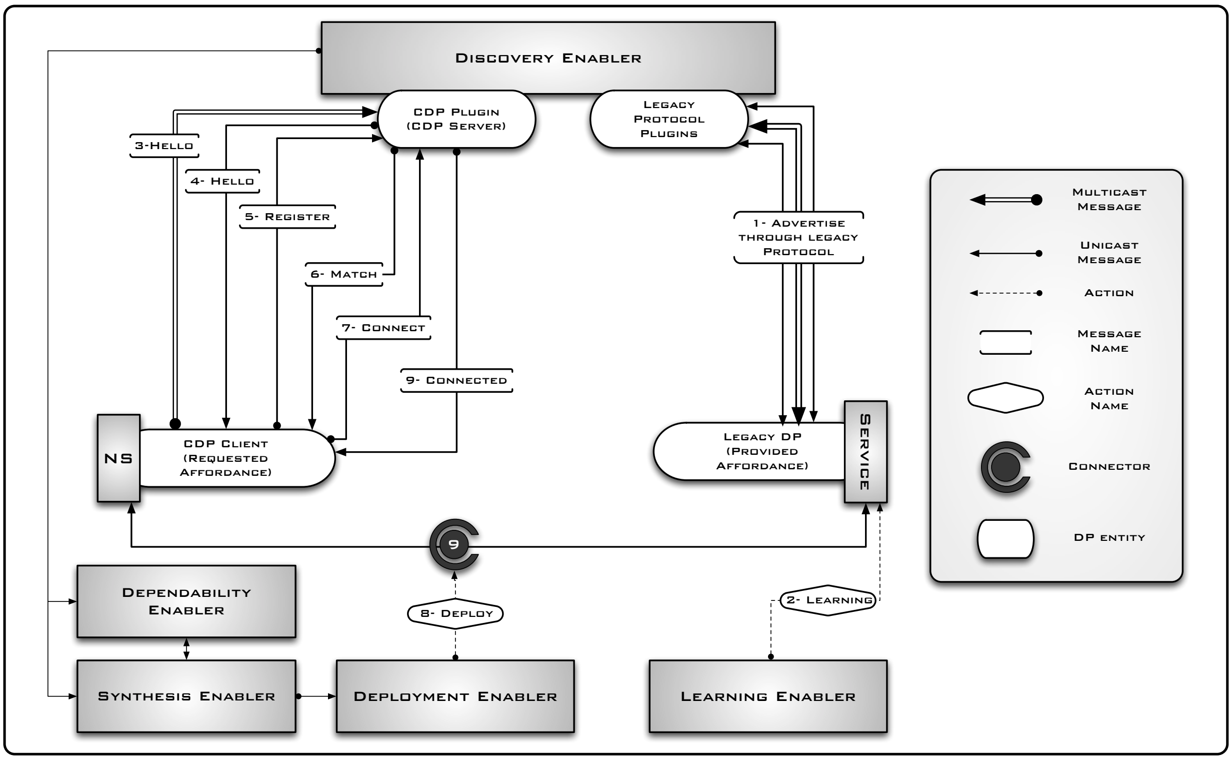


Figure 3.15: Legacy discovery within CONNECT

Figure 3.15 illustrates the discovery of a legacy service that provides some affordance, together with its connection with a CONNECT-aware nodes (i.e., a node using CDP) requiring a matching affordance. First, the relevant legacy plugin discovers the legacy networked system using the embedded discovery protocol (*step 1*). Then, the legacy plugin extracts the base networked system description and further extends it by using semantic inference and soliciting the learning enabler in order to compute the associated NSDL description (*step 2*), which is then stored in the discovery enabler's repository. In the case where a given CDP client (left hand side of the figure) *matches* a legacy networked system, the client receives a *Match* message (*step 4*) and can start interacting with the networked system after it receives a *Connected* message (steps 5-7) in a way similar to the previously discussed CDP-based service discovery. Note that we are currently extending legacy plugins so as to deal with legacy systems requiring *affordances*.

3.4.4 Repository

The repository stores the descriptions of networked system, where we exploit the indexing of networked systems based on their descriptions to include match-based indexing as in [47]. In that work, the repository is structured as a tree in which services with identical *capabilities* are assigned to the same node, and more specific *capabilities* (with respect to the ontology definition) are stored in child nodes. Utilizing that work in our context obviates pair-wise matching ($O(n^2)$) since, if no match is found with a root node, then none of its children need to be checked. However, the repository is still under development, while the current prototype discussed in the next section implements a simple repository.

[13]https://www.eternals.eu/

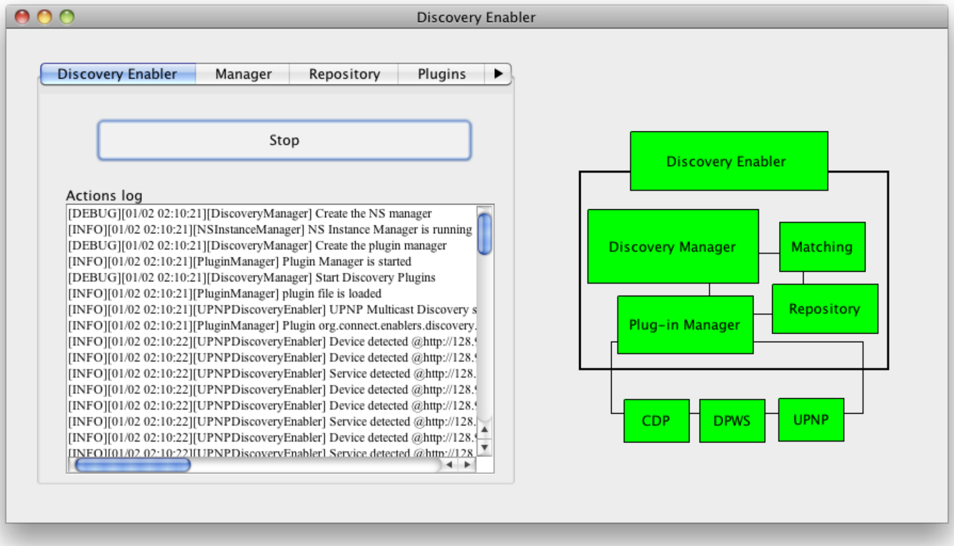


Figure 3.16: The discovery enabler GUI

3.5 Prototype Implementation

The CONNECT discovery enabler is being implemented in Java. According to the design discussed in the previous sections and as depicted on the left-hand side of Figure 3.17, each discovery protocol is encapsulated within a plugin and interacts with the Plugin manager using a uniform internal plugin structure. The set of plugins can be updated and extended dynamically by making changes to a configuration file, which is read by the plugin manager.

At the time of writing, the discovery enabler supports three plugins for the following discovery protocols: the CONNECT Discovery Protocol (CDP), Simple Service Discovery Protocol (SSDP) of Universal Plug-And-Play (UPnP)[14] and WS-Discovery of Devices Profile For Web Services (DPWS)[15]. UPnP is designed for devices on a network to advertise their services and have those services invoked. UPnP includes a discovery protocol (SSDP), which devices use to send advertisements by multicast (over UDP). Each service advertisement contains only an identifier for each service, and so any interested party must request the full description of each service. The returned XML-based service description contains a list of actions that can be performed on each service along with the associated arguments and their types. DPWS is a new emergent standard designed to allow the discovery of networked devices which host Web services. Hence, unlike UPnP, services are described in WSDL and invoked using the normal Web service invocation mechanisms. However, as discussed above, only syntactic interfaces are provided by most legacy discovery protocols. Thus, we have developed an API that provides classes to translate UPnP descriptions, WSDL1.1, WSDL2 and SAWSDL to xDL, while interaction with the other CONNECT enablers allows the NSDL description to be completed. Still, as these enablers are also under development, we have introduced a function, called `ConnectPrim`, which is to be implemented by any legacy system to provide the semantic and behavioral description of its affordances.

Regarding the matchmaking engine, it is implemented according to the design discussed in Section 3.3, using OLTSA to perform behavioral matchmaking.

Figure 3.17 shows the important classes from the JAVA implementation. On the left-hand side are the plugin manager and the three implemented plugins (which delegate to the associated classes for most of the computation involved in handling each protocol). The Matchmaker performs matchmaking by using the HermiT ontology reasoner[16] (which is also used by OLTSA), and the `DiscoveryEnablerGUI` is the

[14] http://www.upnp.org/
[15] http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01
[16] http://hermit-reasoner.com/

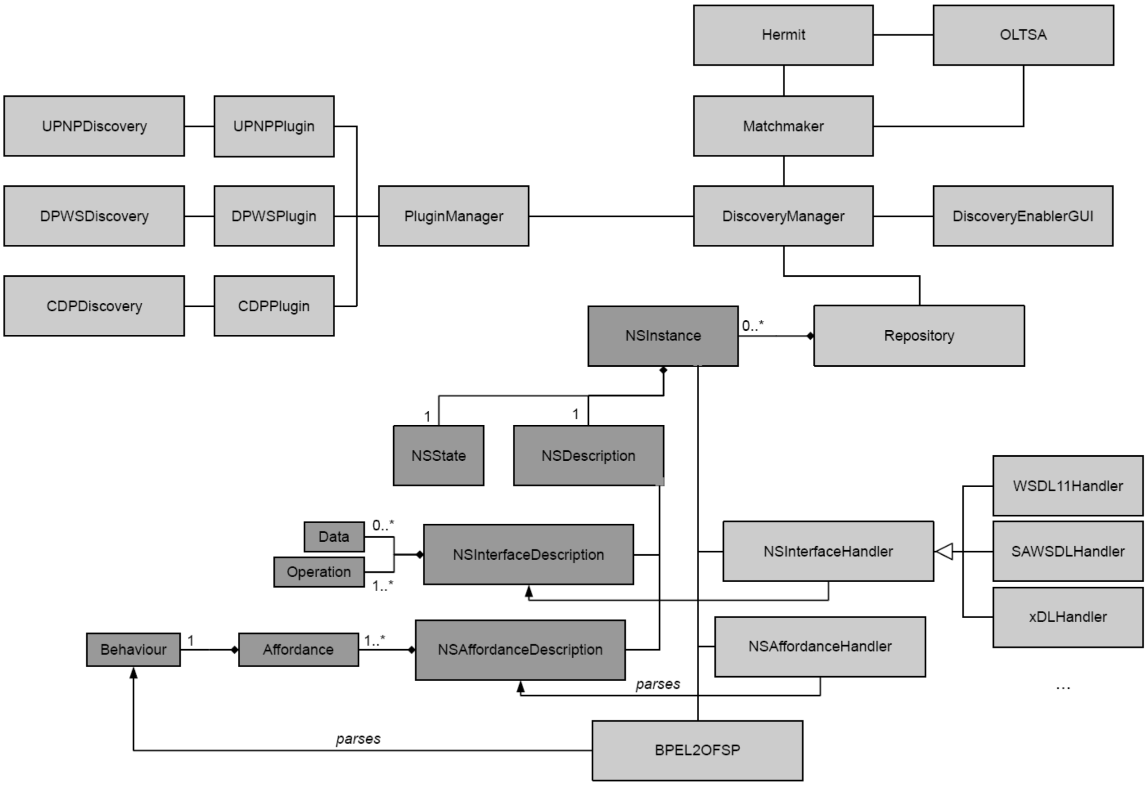


Figure 3.17: Discovery enabler implementation classes. Classes which primarily represent state are shown in dark grey.

root class of the GUI shown in Figure 3.16. The `Repository` keeps a list of `NSInstance` objects which represent an NS with a state and a description, made up of the two parts below it. The `NSInstance` makes calls to a handler to produce each part of the description (normally by parsing a file). This permits handlers for legacy description formats such as WSDL1.1 to be defined.

Our ongoing work relates to enhancing the implementation of the discovery enabler in particular regarding the repository so as to enhance the performance of the discovery process. Also, the matchmaking implementation will evolve according to the findings of our work on CONNECTor synthesis in WP3, and in particular the advanced mediation that is targeted. Furthermore, integration with the other enablers is under way so as to make the CONNECTion of networked systems seamless. However, the discovery enabler may be used in isolation regarding provided support for universal discovery. Another part of our ongoing work is to conduct extensive experiments so as to assess the current design and in particular identify performance bottleneck, including with respect to the use of advanced reasoning tools that significantly impact performance. In this direction, evolution of the repository to enable match-based indexing is a promising approach. Finally, note that in order to be able to perform extensive testing of the Discovery Enabler, we are currently collecting networked system model instances.

3.6 Conclusion

The CONNECT discovery enabler is a key element of the CONNECT architecture as it initiates CONNECTion between networked systems, i.e., it enables identifying pairs of networked systems that are behaviorally compatible and may thus be CONNECTed through a supporting CONNECTOR. Building on the extensive work in the area of interoperable and semantic service discovery protocols, including by the CONNECT consortium partners, this section has introduced the base design and early prototype implementation of the CONNECT discovery enabler. Compared to the relevant state of the art, the proposed enabler distinguishes itself by dealing with the discovery and behavioral matchmaking of highly heterogeneous networked systems implementing different coordination paradigms. The proposed solution to behavioral matchmaking relies on the mapping of communication paradigms to base input and output communication actions, according to the matching and mapping relations introduced in Deliverable D3.2 [18]. Furthermore, we have introduced an extension to the LTSA tool to support automated behavioral matchmaking according to both the ontology-based semantics of communication actions and the behavior of networked systems.

Our ongoing work in the area relates to

- further integration of the implementation with the other CONNECT enablers and in particular the learning and synthesis enablers,
- enhancement of the NS repository to use efficient methods of storing NS descriptions by exploiting the matching relationships between affordances and the other components of the complete description, and
- experimental evaluation so that the enabler provides adequate performance.

A more challenging area of future work is in the inference of semantic information from natural-language content—like method names and documentation comments—in legacy interface description languages. Through these means, affordances may be automatically extracted, and interface descriptions annotated with ontological references. This will ultimately enable transparent CONNECTion of systems using discovery protocols that provide only syntactic descriptions.

4 Realising CONNECTors

4.1 Introduction

The key objective of CONNECT is to produce CONNECTors. In this section we discuss in more detail what this entails in terms of realising the CONNECTor architecture (illustrated in Figure 2.4) that was described in Section 2.4. In particular we focus on how the two important elements of that architecture, have been designed and implemented (or in some cases, generated):

- We describe how Domain Specific Language (DSL) descriptions of protocols are used to generate the *listeners* and *actuators* that perform the important role of communicating with networked systems using the legacy protocol that these employ.
- Three alternative designs are then presented for the implementation of mediators. These include the code-generation approaches as produced by WP3 [18], and also model execution approaches where the specification of the mediator is executed directly.

To evaluate the CONNECTor architecture in practice we provide two case studies. The first demonstrates how CONNECTors are generated dynamically in order to allow three discovery protocols to interoperate. Discovery protocols were chosen for this case study for two reasons: i) they are simple middleware protocols that can interoperate without consideration of the application heterogeneity, and hence offer a good first case for full CONNECTor generation; ii) these protocols are also used within the discovery enabler, hence, we consider the approaches presented to potentially generate the plug-ins used in the discovery enabler. The second case study investigates a more application-oriented system, namely multiple instant messaging systems that use heterogeneous protocols (e.g. MSN, Yahoo, XMPP IM protocols). Here we show how CONNECTors can be created to ensure they interoperate; at this stage these CONNECTors are hand-crafted but we are working towards dynamically generating equivalent software.

4.2 Realising Listeners and Actuators

4.2.1 Motivation

CONNECTors employ a simple mechanism to communicate with legacy protocols. Listeners read messages sent by legacy protocols (these are received as an array of bytes from the network engine) and then transform these to the *Abstract Message representation*. Actuators do the reverse to create legacy protocol messages as a byte array and transmit these using the network engine. Listeners and Actuators can be implemented using a number of different approaches. We now discuss the benefits and drawbacks of three potential approaches to implement the Listeners and Actuators within the CONNECT project.

1. *Third party middleware wrappers*. The middleware binary or library (as implemented by the original third party developers, i.e., not developed by CONNECT) is directly used by listeners and actuators; these wrap the middleware code by interfacing with the API provided by the binaries.
2. *Connect Specific Plug-ins: Manual Coded*. CONNECT developers implement the required listener and actuator code for a given protocol. That is, they hand-code the extraction of information from network packets and then translate to the Abstract Message. Note, this is the approach that is currently employed to implement the plug-ins that are utilised by the discovery enabler.
3. *Connect Specific Plug-ins: Automatically Generated*. Listeners and actuators are generated and deployed automatically; each protocol's behaviour(message sequence) and message format (packet format) is specified using high-level models. The models are then used to generate the required software.

Third party middleware wrappers

The benefits of this approach are:

- Reduced programmer effort. The middleware code does not need to be re-implemented, only the wrapping to the CONNECT software elements.

The drawbacks of this approach are:

- Duplicated implementation. Each middleware will naturally duplicate implementation found in every other middleware, e.g., code for network transport, code for marshalling, etc; this will unnecessarily add to the size of listeners and actuators.
- Unnecessary functionality. A CONNECTor only requires the ability to parse and compose network messages. However, the middleware binary will likely contain significant other features which will add to the deployment size of the CONNECTor.
- Complex configuration. Each middleware binary will have its own installation and configuration routine and hence, heterogeneous deployment strategies will need to be devised and executed.
- Wrapper development is complex. Writing wrappers may involve small pieces of code, but these will be complex to develop, as they require understanding of the middleware API, the CONNECTor process, and the relationship between the two. The mappings will also have to consider variations in programming abstractions, e.g., synchronous and asynchronous middleware APIs mapped onto the listeners and actuators.

Connect Specific Plug-ins: Manual Coded

The benefits of this approach are:

- Complete control of configuration and deployment of middleware behaviour within CONNECT. The software is implemented to be easily configurable and deployable within a CONNECTor.
- Required functionality only. CONNECT developers will implement only the necessary middleware code and hence, the deployment code will be optimised for the CONNECTor process and save resources.

The drawbacks of this approach are:

- Significant development costs for each plug-in. The code to receive, parse, compose and send messages for each protocol must be individually developed; this must then be integrated into Listeners and Actuators. This involves the complex task of understanding each protocol and breaking down its operation to fit within CONNECT.
- Reduces the possibility of being carried out by 3rd party developers. To carry out this implementation, a strong understanding of how CONNECTors operate and are implemented is required; this may limit the development to CONNECT developers only, and hence limit the number of protocols that can feasibly be developed with the project resources.
- Duplicated behaviour. The implementations that are carried out independently by different developers may still duplicate some middleware functionality (e.g. network transport) leading to unnecessary resource consumption.

Connect Specific Plug-ins: Automatically Generated

The benefits of this approach are:

- Reduced development costs as only the high-level specifications are required. These can also be supplied by third-party developers without understanding how CONNECTors are implemented.

- The automation process produces the required middleware functionality only; further, there is no duplicated middleware functionality because the code generation approach is built upon re-usable components.
- Potential for high-level specifications to be learned. Learning enablers in the CONNECT process generally produce high-level models that can then be used to automatically generate software; this may also be the case for the important Listeners and Actuators within the architecture.

Any of the three approaches would provide an acceptable implementation for the CONNECTors (indeed, due to the low level nature of implementation it might go unnoticed). However, based upon the potential benefits we decided to undertake the **Connect Specific Plug-ins: Automatically Generated** approach. The reduction in development costs, and the potential for the solution to reduce the resource usage costs of CONNECTors were important factors. However, the deciding factor was that this approach is the most likely to be future-proof, and as such fit with the general goals of the CONNECT project. The ability to learn high-level models and then generate the corresponding code from it applies the CONNECT ideology to the lowest level of implementation.

4.2.2 Message Description Languages

The general philosophy employed for the deployment of Listeners and Actuators is to utilise DSLs to describe protocol messages. These high-level descriptions are then used to create the software components that will be deployed in the CONNECTors. A Message Description Language (MDL) is the language used to describe a message format; the MDL specification for a particular protocol then describes its set of messages only. Message composers and parsers are implemented as general interpreters that execute the message description language specifications that are loaded. For example, a parser that interprets an SLP MDL instance will parse SLP messages into the abstract message representation, i.e., it interprets the incoming message based upon the specification. Hence parsers are specialised to a particular protocol by associating the protocol specification to produce the Listener. Actuators are created using the same process to specialise generic message composers for text and binary protocols. An overview of this process is illustrated in Figure 4.1.

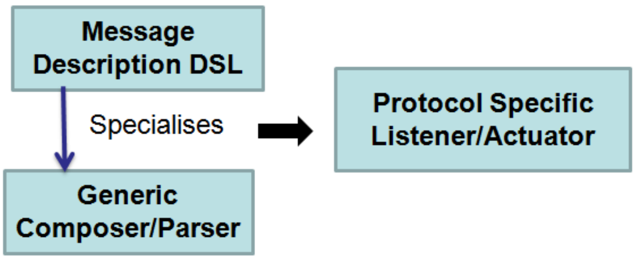


Figure 4.1: The approach for generating Listeners and Actuators

There are a number of languages that can be used to parse network messages, or parse data files. We investigated each of these as potential languages to be used in CONNECT; the results of this are seen in Figure 4.2. It can be seen that a number of the tools focus solely on generating software to parse only data and messages, i.e., BinPac, Datascript and PacketTypes; therefore, these are unsuitable as it is equally important to be able to generate the composer part of the CONNECTor. Similarly, a number of the languages only consider binary data (i.e., all except PADS and ASN1.0); however, CONNECT requires the parsing of heterogeneous protocols which may use text or XML. Hence, the only potential solutions are: i) PADS which offers the additional benefit of being able to infer data descriptions from received data [24], or ii)ASN 1.0. The drawback of these two are that they are not specifically designed for network packets, and we found when creating descriptions of example packet formats for SLP and GIOP that we were unable to successfully create the correct parsers and composers. Given the results of this investigation, we decided to develop our own MDLs and corresponding tools in order to first provide a simple mechanism to parse and compose network packets, and also allow us to easily extend the language as we encounter heterogeneous protocols.

Tool	Langauge	Generate Parser	Generate Composer	Domain
ASN 1.0 [63]	Java/C	x	x	Many data encodings: binary, text, xml
BinPAC [54]	C++	x		Binary data and network packets
Datascript [3]	Java	x		Binary data
PADS [23]	C/ML	x	x	Binary or Text data
PacketTypes [44]	ML	x		Binary network packets
Melange [40]	ML	x	x	Binary network packets

Figure 4.2: Comparison of Data and Message Description Languages

CONNECT is flexible to allow different types of language to be used to specify message formats; each language can be termed an MDL. This flexibility better supports the parsing and composing of a wide range of protocols. For example, specialised languages for binary messages, text messages and XML messages can be utilised. To illustrate the approach we present a language for binary messages, and then a language for text messages. It is important to identify here that the role of these languages is to extract the information into a representation that is usable within CONNECT; the languages themselves do not seek to understand the content of the message, nor are they concerned with the application semantics of the message. Take for example an RPC request message invoking an operation `Foo`, these languages can extract the value 'Foo' for the label 'operation' but cannot determine its purpose. In Section 6, we examine the role of ontologies in understanding labels and the application content such that this can be used to address: data, application, and middleware heterogeneity; thus going futher than tackling middleware heterogeneity as presented in this section.

4.2.3 Binary MDL

We use examples to succinctly illustrate the features of the language and how it is used in practice. Figure 4.3 shows part of the MDL specification for the SLP protocol (a binary protocol). In the specification there are three important constructs:

- `<Types>` list the types of each individual field type, e.g., the `Version` field type is an integer value. Types are separated from the message specification in order for field types to be reusable across multiple messages.
- `<Header>` includes the message format of the header for the binary protocol messages. If a header specification is present this is common to every message in the protocol (only one Header can be defined).
- `<Message>` describes the packet format for the body of a particular message. Each protocol will typically contain multiple message bodies, for example the SLP protocol here contains message bodies for two messages: an SLP request message, and an SLP reply message.

Hence, `<Header>` and `<Message>` specify the content of the message headers and bodies. These are both composed of `<label:size>` entries for each field in the message. The *size* is the length of the field content in bits. There is one special label: `<rule:field=value>`. This is used to relate the correct message body with the header. For example, the SLP `SrvReq` message applies when the value of the `FunctionID` field in the header equals one.

To underpin the reading and writing of data from messages, pluggable marshallers (a method prefixed with 'out') and unmarshallers (a method prefixed with 'in') for each of the types are utilised. For example, Integer has a plug-in marshaller that writes Java Integers to a byte array, and a corresponding unmarshaller that transforms a byte array to a Java Integer. When the "Integer" value is found as the type of a field it executes the `inInteger` method to read values from the byte array and the `outInteger` method to write values to a byte array. This mechanism allows the language to be dynamically extended to incorporate complex types (with no need to re-implement a compiler) that may be specific to a particular protocol. For example the `FQDN` type is a format for domain names (Fully Qualified Domain Name) that is employed by the DNS protocol; to add the `FQDN` type to this language, we simply plug-in marshallers

```
<Types>
    <Version : Integer><Function−ID : Integer>
    <MessageLength : Integer [ f−MsgLength]>
    <Next−Ext−Offset : Integer><XID : Integer>
    <SRVTypeLength : Integer><SRVType : String>
    <URLLength : Integer [ f−length (URLEntry)]>
    <URLEntry : String>
<EndTypes>

<Header : SLP>
    <Version:8><Function−ID:8><MessageLength:24><reserved:16>
    <Next−Ext−Offset:24><XID:16><LanguageTagLen:16><Language−Tag : LanguageTagLen>
<End : Header>

<Message : SrvReq>
    <Rule : Function−ID=1><PRLength:16><PRString : PRLength>
    <SRVTypeLength:16><SRVType : SRVListLength><PredLength:16>
    <Predstring : PredLength><SPILength:16><SPIstring : SPILength>
<End : Message>

<Message : SLPSrvReply>
    <Rule : Function−ID=2><ErrorCode:16><URLEntryCount:16>
    <URLEntry : URLEntryCount>
<End : Message>
```

Figure 4.3: Partial view of the SLP message description

that map `FQDN` byte arrays to a Java String. The marshaller and unmarshaller for `FQDN` are illustrated in Figure 4.4. This feature provides us with an advantage over the alternative DSL methods (e.g. PADS) in Figure 4.2 where extending the type system is complex and unwieldy and cannot be done at runtime.

Another feature of the `<Types>` specifications are functions. Functions can be defined on types using the `[f-method()]` construct e.g. `[f-length(URLEntry)]` in Figure 4.3. They are generally useful for calculating values that must be composed when creating a message (rather than parsing), i.e., the named f-method is executed by the marshaller to get the value that must be written. Importantly the function can take other message fields as parameters, e.g., to compose the value of the `URLLength` field, you need to obtain the length of the URLEntry field. Hence, the marshaller takes the value of the `URLEntry` field, calculates the length and then composes this as the `URLLength` field value.

4.2.4 Text MDL

Text based protocols are different from binary protocols and therefore, a new MDL is required to generate the Listeners and Actuators. We again use an example specification to highlight the features of the Text MDL; a subset of the messages within HTTP is specified in Figure 4.5. Like the binary approach there is a list of field labels with their corresponding types in the `<Types>` section and again `<Header>` and `<Body>` are used to describe the individual messages. The key difference in this language is that we utilise field deliminators rather than bit lengths to distinguish the length of the fields. For example in the `<Header>`, `<Method:32>` means that the field is terminated by the '32' ASCII character, i.e., a space. In the case where multiple characters are used to delimit we employ commas to list the character values e.g. `<Version:13,10>` is a backslash r followed by a backslash n.

Another important feature of text protocols is that they are typically self-describing, i.e., the field label as well as the value will form the content of the message. For example, a HTTP message may contain "Host:www.lancs.ac.uk"; this defines a field with a label Host and a value www.lancs.ac.uk. Hence, text protocols are not rigidly defined in terms of the fields and their order. To support this property we employ the <**Fields:** > construct; this will parse/compose a list of free form self-describing fields into their label, size and values. For example, `<Fields:13,10:58>` splits fields using the 13,10 delimitor, then it uses the 58 value (a colon) to split the field into its label and value. The label must relate to a type specified in the `<Types>` section.

```java
/**
 * Covert a byte array of FDQN format to a Java String. FDQN is where
 * a zero terminated byte array contains the number of letters between '.'s
 * e.g. 3www5lancs2ac2uk0 parsed to "www.lancs.ac.uk"
 */
public String inFDQN(byte[] bytes) throws DataNotOfTypeException{
    String query="";
    int index=0;
    while (index<bytes.length){
        int portionLength = bytes[index++];
        byte[] portion = new byte[portionLength];
        System.arraycopy(bytes, index, portion, 0, portionLength);
        index+=portionLength;
        query+=(new String(portion));
        if(index<bytes.length)
            query+=".";
    }
    return query;
}

public byte[] outFDQN(String Value, int Size){
    StringTokenizer st = new StringTokenizer(Value, ".");
    byte[] result = new byte[Value.length()+2];
    int index=0;
    while (st.hasMoreTokens()){
        String delim = st.nextToken();
        result[index++] = (byte) delim.length();
        byte[] delimBytes = delim.getBytes();
        for(int j=0;j<delimBytes.length;j++){
            result[index++]=delimBytes[j];
        }
    }
    result[Value.length()+1]=0;
    return result;
}
```

Figure 4.4: Java implementation of plugabble marshalling and unmarshalling methods for FQDN fields

4.3 Mediators

We are currently investigating three different approaches to implement mediators, which perform the central translation role within a CONNECTor; effectively these co-ordinate the required behaviour between the Listeners and Actuators. These three approaches are as follows:

- *Code generation*. The complete mediator behaviour is synthesized from the Networked System Model. There is no need for a mediator engine because all functionality is included in the synthesis.
- *BPEL based mediators*. The mediator behaviour is modelled using the BPEL co-ordination language; the corresponding mediator engine is a BPEL engine that executes the BPEL scripts.
- *Interpretation of LTS models*. We extend the LTS model with: i) colours (to describe how the mediator interacts with the network), and ii) message translations to describe how fields from one message of a protocol are translated into the corresponding message(s) of the protocol to be interoperated with.

4.3.1 Code Generation

The objective of code generation is to build an *ad-hoc* CONNECTor that can be directly deployed and run. The code is generated from the CONNECTor model outputted by the synthesis. This model is an LTS and is first translated into a Mealy machine where each transition is labelled with a trigger event and a list of

```
<Types>
      <Method : String ><URI : String ><Version : String >
      <From : String ><Host : String ><Referer : String >
      <TE : String ><User-Agent : String ><CONTENT-TYPE : String >
      <CONTENT-LENGTH : String ><DATE : String >
       ...
      <Body : Octets>
<EndTypes>

<Header : HTTP>
      <Method:32><URI:32><Version:13,10>
      <Fields:13,10:58>
<End : Message>

<Message : HTTP_GET>
   <Rule : Method=GET>
<End : Message>

<Message : HTTP_Response>
      <Rule : Method=HTTP/1.0>
      <Body : CONTENT-LENGTH>
<End : Message>
```

Figure 4.5: Partial view of the HTTP message description

actions to perform. The resulting Mealy machine can be seen as a function which selects the actions to execute with respect to the current state of the automaton and with respect to the last event that occurred.

One of the simplest way to implement a Mealy machine is to use the nested switch strategy, which advocates the use of two nested conditional branching statements: the first one selecting the current state of the machine, and the second one selecting the current message. The following code excerpt illustrates this strategy:

```java
while (!stopped) {
  Message message = getLastMessage();
  switch(currentState) {
  case State1:
    switch(message.getType()) {
    case Msg1:
      // Perform the actions triggered by Msg1 received in State 1
      break;
    case Msg2:
      // Perform the actions triggered by Msg2 received in State 1
      break;
    default:
      // Illegal message in state 1
    }
    break;
  case State2:
    switch(message.getType()){
      // idem than for state 1
    }
    break;
  default:
    // Illegal state
    break;
  }
}
```

A more complete description of the code generation process, which further motivates the use of the nested switch pattern can be found in the Deliverable D3.2 [18].

The code generation does not only output some Java source files, but it also outputs two additional resources needed to compile and package the CONNECTor into an OSGi bundle: the manifest file and the build file. The Manifest file explicitlt identifies the dependencies that the CONNECTor may have on other bundles, and also contains other information needed to properly package the CONNECTor. The *Ant Build* *file* describes the rest of compilation process, and specifies how to compile the Java source files, how to wrap them into a Jar file compliant with the OSGi platform, and where to move that file so the CONNECTor can properly start.

The code generator has been implemented using the JET toolkit (Java Emitter Templates). JET enables the definition of code templates that can be used to specify the code generators. A template represents the piece of source code that is expected; this consists of the parts that may vary depending on the model, which will be replaced by some Java code. The code generators which produce the different artifacts result from separated templates.

4.3.2 BPEL

One alternative to the generation of an *ad hoc* connector is the generation of a service orchestration. Services orchestrations are similar to CONNECTors, since they both aim at coordinating remote entities that communicate by exchanging messages. The main advantage of the generation of a service orchestration is the gain in flexibility which comes from the fact that service orchestration models are models that are dynamically executed.

The *defacto* standard used to implement service orchestrations by the both academics and industry is the BPEL language. BPEL is a block structured language whose programs are sequences of statements including remote service invocation, conditional branching, data manipulation, etc. However, BPEL is based on the assumption that all the services involved in the orchestration are homogeneous: they all exchange SOAP requests to communicate. From this perspective, CONNECTors may be seen as a generalization of service orchestrations, made to compose heterogeneous services. The use of BPEL to capture such compositions of heterogeneous services implies however to extend the behavior of the underlying execution engine (such Active BPEL or Apache ODE) in order to enable communication with versatile middleware technologies. This will be investigated in the third year of the CONNECT project.

Since BPEL is a block structured language, including most of the statements that are commonly found in imperative languages such as C or Java, it is therefore possible to reuse the same code generation strategy to obtain a BPEL orchestration. The code generator has been hence also implemented using the JET toolkit and merely outputs a BPEL file, which mimics the behavior of the LTS using the nested switch statements strategy aforementioned.

4.3.3 Interpretation of LTS Models

The key philosophy behind this mediation approach is to interpret the high-level models as produced by CONNECT directly. However, the LTSs require extensions in order to interoperate at a low level with legacy protocols. These extensions are as follows:

- *k-Coloured LTS* The behavior of a protocol is described by an LTS where transitions represent message exchanges. However, protocols vary in their interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. We introduce the concept of k-coloured LTS to capture the network properties of a protocol by a colour k.

- *Merged LTS.* When several protocols need to interoperate, it is necessary to express the relation among them and to describe the message translation logic, which defines how to translate messages from one protocol to another. Protocol interoperability is defined in a merged LTS that describes how to combine the k-coloured LTS of the protocols involved and produce the final mediator to be interpreted.

- *Translation Logic.* The role of the translation logic is to describe the translation of data and behaviour where messages are semantically equivalent. One key operator of the language is the *assignment operation*. Assignment allows the content of one or more fields of a particular message, to be translated to the fields of a different message.

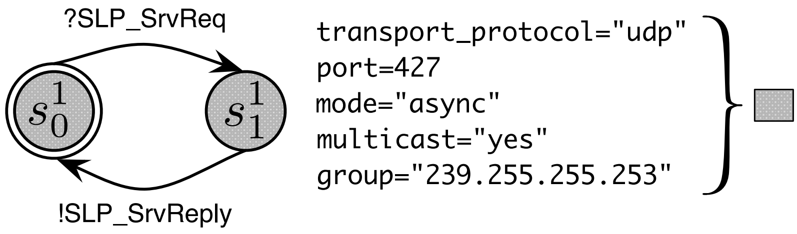


Figure 4.6: SLP coloured LTS

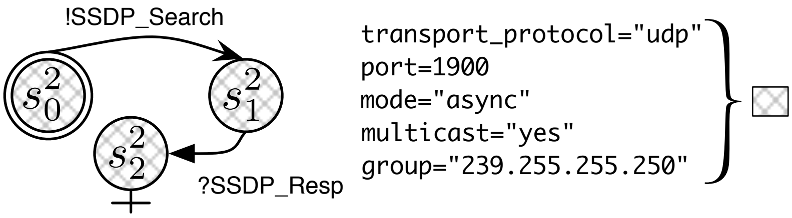


Figure 4.7: SSDP coloured LTS

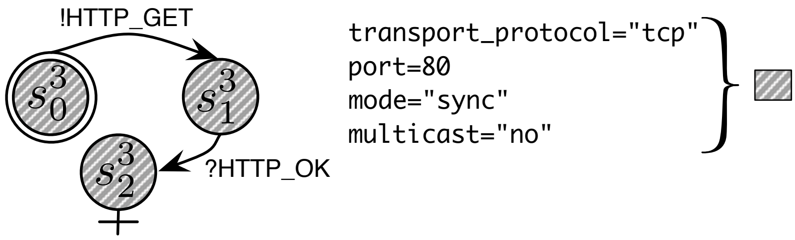


Figure 4.8: HTTP coloured LTS

$k-$Coloured LTS

Acknowledgement. We acknowledge the collaboration between Lancaster University and the University of Bordeaux (specifically David Bromberg and Laurent Réveillère) in creating and implementing the ideas of $k-$Coloured LTS.

Protocols may not only differ in their behavior but also in the the way they use the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. For instance, as illustrated in Figure 4.6 and Figure 4.7, although SLP and SSDP protocols are both asynchronous and multicasted, they differ from their multicast address and port number which are $239.255.255.253 : 427$ for SLP and $239.255.255.250 : 1900$ for SSDP. Note that, sent messages are not necessary received asynchronously, it depends of the underlying network details. In order to capture these low level network semantics, we use colouring which consists of assigning labels called colours to states of the LTS. An LTS can pass successfully from one state to another, following either a *receive-transition* or a *send-transition*, without any network issues, only if the concerned states share the same colour. For instance, as described in Figures $\{$4.6, 4.7, 4.8$\}$, according to their transport protocol, port, mode, multicast and group attributes, a specific and different colour has been affected for the SLP, SSDP, and HTTP LTS.

Merged LTS

Interoperability requires that when one legacy system $\mathcal{L}_1$, relying on a protocol $\mathcal{P}_1$, sends a sequence of messages $\vec{m}$ to another legacy system $\mathcal{L}_2$ that relies on a different protocol $\mathcal{P}_2$, then $\mathcal{L}_2$ must be willing to receive these messages after a set of transformations to resolve mismatch issues at both the network layer and at the message layer in terms of message format, content and sequence. However, there is

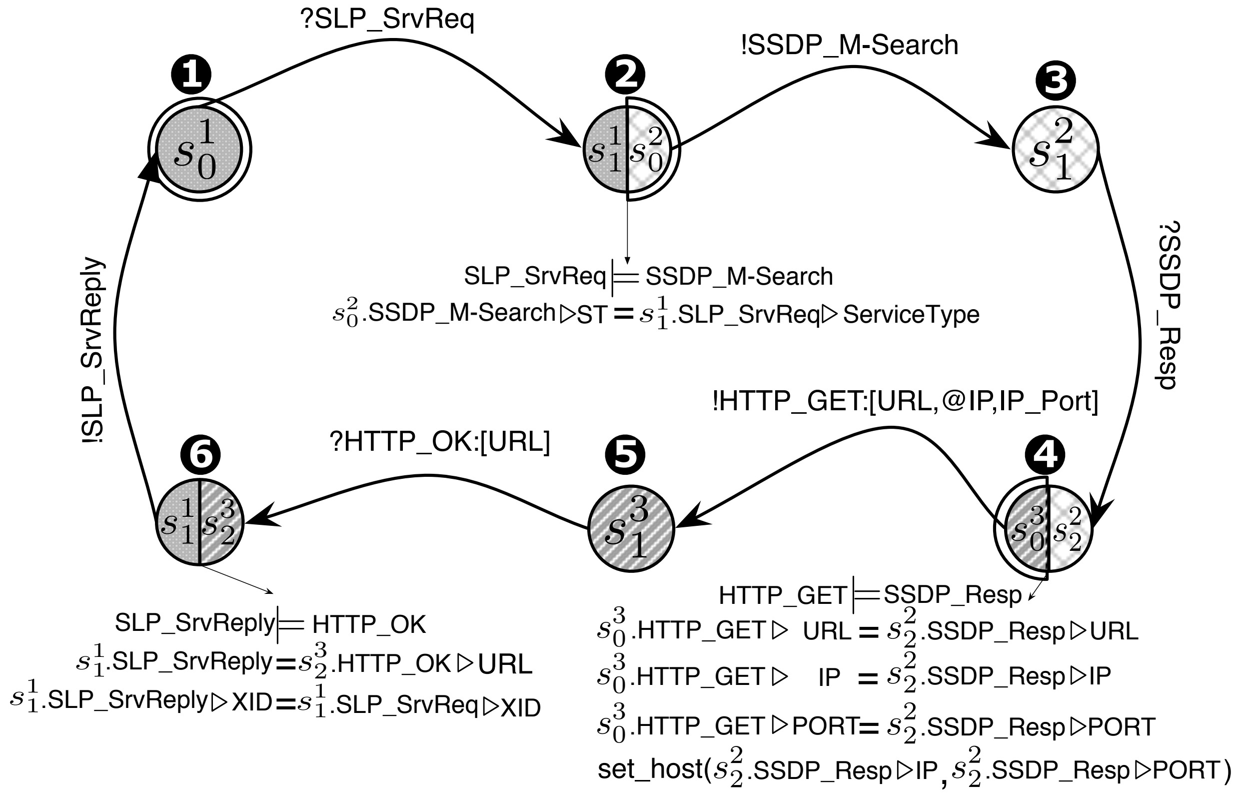


Figure 4.9: A merged k-Coloured LTS for SLP, SSDP and HTTP protocols

a prerequisite to successfully apply these transformations: there is a need to reason about the meaning of messages that are willing to be transformed. For instance, it appears from Figure 4.6, and Figure 4.7 that a SSDP service should be able to understand what kind of services are requested by a SLP client if at least a `SLP_SrvReq` request is semantically equivalent to a `SSDP_Search` request. Still, a semantic equivalence is not always as simple as a one to one mapping among two messages. In the most general use case, there are several patterns of semantic mismatches hampering interoperability. A protocol $\mathcal{P}_1$ may require a single message to perform a particular task, while in another protocol $\mathcal{P}_2$, a similar task is performed by receiving several messages. Alternatively, another type of mismatch occurs if the protocol $\mathcal{P}_1$ needs to receive several messages to perform a particular task, while in the protocol $\mathcal{P}_2$ the same task is achieved with only one message.

Intuitively, protocols are interoperable if there is a way to merge their respective coloured LTS. Figure 4.9 illustrates a merged LTS for SLP, SSDP and HTTP. States that are linked by a $\delta-$*transition* are represented by bicoloured nodes such as nodes ❷, ❹ and ❻. The initial state of each LTS, which are expected to be merged, are reached by a $\delta-$*transition*. Hence, a SSDP `M_SEARCH` is sent as soon as a SLP `SrvReq` sent by a client is received. Then a HTTP `GET` request is sent at the reception of a SSDP `Resp` message. Finally we send back to the SLP client a SLP `SrvReply` after having received a HTTP `OK` response. In other terms, $\delta-$*transitions* enable to chain SLP, SSDP and HTTP LTS through a directed path that both starts and ends in the same LTS.

Translation Logic

Figure 4.9 shows examples of how the translation logic is applied at the bridging states between two protocols. At node ❷ (the first bridge between SLP and SSDP), as `SrvReq` $\models$ `M-Search`, we assign the `ST` field of the SSDP `M_SEARCH` message with the `ServiceType` field from the received SLP `SrvReq` message. Similarly, at node ❻, we assign the resulting URL from the received `HTTP_OK` message to the URL field of the SLP `SrvReply` message to be sent.

$\delta-$*transitions* are also used to define additional behaviour required by the LTS that is guided by the content of messages. For example, in order to connect to a HTTP server to perform a GET message request in Figure 4.9 we need to know the address and port. However, this information is only obtained from the content of the `SSDP_Resp` message. Thus, node ❹ uses a keyword operator *setHost* that takes the fields host and port from the message and sends them to the underlying network engine to point the next TCP connection.

The Mediator Engine

The Mediator Engine, like the listeners and actuators, interprets the merged LTS. This engine is implemented to read these models from XML content (the merged k-Coloured LTS is produced in XML). To give a flavour of the concrete operation of the framework we briefly summarise the behaviour that occurs at the different state types: *receiving, sending, and bridge*.

At a **receiving state** $R1$, the mediator engine listens for messages using the network engine for the protocol address and port (of the state); when a message is received it is parsed by the listener. If the abstract message's name label matches one of the transition labels then the automata moves to the pointed to state $S1$, and then pushes the `Abstract Message` onto the message queue at $R1$.

At a **sending state** $S1$, the mediator engine reads the label of the transition and then constructs this message using a message composer before using the network engine to send it correctly with the required network transport semantics of the protocol. In the case where content has been translated by a prior state, the state $S1$ retrieves the message to be sent from the queue of a prior state before composing and sending.

A **bridge state** $B1$, represents an intermediary state from the bi-coloured states (e.g. ❷, ❹ and ❻ in Fig. 4.9). These states do not send and receive messages, they only translate content from one abstract message to another or perform logic required to underpin interoperability. The XML content in Figure 4.10 shows an example of such a state (for the SLP `SrvReq` to SSDP `M-SEARCH` translation), presenting only the translation logic. For field assignments, the engine reads the value from the second field (as pointed to by the XPath expression), this equates to reading the value from the Java object of the abstract message, and then writes the content to the abstract message whose field is pointed to by the first field node in the XML <Assignment> content.

```
<Bridge>
    . . .
    <TranslationLogic>
      <Assignment>
          <Field>
              <Message>SSDP Search</Message>
              <Xpath>/field/primitiveField[label= ST ]/value</Xpath>
          </Field>
          <Field>
              <Message>SLPSrvRequest</Message>
              <Xpath>/field/primitiveField[label=SRVType ]/value</Xpath>
          </Field>
      </Assignment>
    </TranslationLogic>
    . . .
</Bridge>
```

Figure 4.10: Translation logic expressed in XML

4.4 Creating a Prototype CONNECTor: Interoperation between SLP, UPnP and Bonjour

4.4.1 Goals

The overall goal of this experiment is to achieve interoperability between three protocols that are heterogeneous in terms of their behaviour and message format, but perform similar tasks (this offers an ideal first case due to the semantic similarities between the protocols). The three protocols selected are: SLP, UPnP and Bonjour which all perform the discovery and advertisement of services. We hypothesize that we can create a CONNECTor for each protocol pair using only high-level models of these communication protocols, i.e., there is no implementation or deployment of legacy code that is specific to the behaviour of an individual protocol. Not to be confused with the previously documented discovery enabler, this is a separate self-contained experiment that seeks to highlight that we can achieve protocol interoperability; however, there is also the potential of this experiment to have wider impact across the project, i.e., that the results themselves can be used to broaden the intoperability of the discovery enabler. In the enabler, only a couple of protocols have been implemented as plug-ins; generating these plug-ins from high-level models may offer a good approach to populate the enabler behaviour.

The overall objective of this experiment is to develop the CONNECT models for each of the protocols such that we can take the legacy applications implemented upon the three protocols and ensure they can interoperate with one another, i.e., that an SLP application's lookup request can be answered by either a UPnP service or a Bonjour Service by deploying the CONNECTor in the network. There are six particular cases: SLP to UPnP and Bonjour, UPnP to SLP and Bonjour, and Bonjour to SLP and UPnP. For conciseness, we discuss only two cases in detail:

- *SLP to Bonjour*. These two protocols are both binary protocols and their message sequences are similar. They differ in message content and network addresses.
- *SLP to UPnP*. In this case, there is heterogeneity of the protocol messages and the behaviour message sequence. SLP employs binary messages, while UPnP uses text-based messages. SLP is a simple request response, whereas UPnP involves multiple requests to the service.

4.4.2 Methodology

The first step of the experiment was to develop simple legacy applications to lookup a simple test service, and respond to lookup requests for the simple service. For SLP we used the OpenSLP protocol implementation[1]; for UPnP we used the Cyberlink Java implementation[2]; and for Bonjour we employed the Apple Bonjour SDK for Windows[3].

We chose to develop CONNECTors which use the coloured LTS mediator engine (as described in Section 4.3.3).

SLP to Bonjour

For the SLP to Bonjour case, we created five different specifications that are loaded into the CONNECT deployment enabler: i) an MDL specification of SLP messages as previously illustrated in Figure 4.3, ii) an MDL specification of Bonjour messages, as illustrated in Figure 4.11 (Bonjour uses DNS messages so this MDL describes DNS questions and responses), iii) a coloured LTS of SLP (see Figure 4.6), iv) a coloured LTS of Bonjour as shown in Figure 4.12, and v) a merged LTS as shown in Figure 4.13.

SLP to UPnP

UPnP utilises two protocols to perform service discovery: SSDP messages are sent with the original lookup request, an SSDP response gives information about a device hosting that service. A further HTTP request is then needed to retrieve the URL of the service from this device. Hence, in this case seven models were loaded into the framework: i) the SLP MDL, ii) the SLP coloured automaton, iii) the SSDP MDL, iv) the SSDP coloured LTS, v) the HTTP MDL, vi) the HTTP coloured LTS, and vii) the merged LTS for the three protocols. An important difference here is that SSDP and HTTP are text messages and as such require a different type of MDL and corresponding parser and composer. Fig. 4.14 shows the content of the SSDP MDL, this identifies the general boundaries of fields "e.g.\r\n" (chars 13,10) because there is no fixed layout or ordering of fields. The inner field boundary (e.g. the ':' split - char 58) then takes the field label from the left and the field value from the right to build a field in the abstract message. When the SLP

[1] http://www.openslp.org/
[2] http://www.cybergarage.org
[3] http://developer.apple.com/opensource/

```
<Types>
    <ID : Integer><QR : Boolean><Opcode : Integer><AA : Boolean><TC : Boolean>
    <RD : Boolean><RA : Boolean><Z : Integer><Rcode : Integer>
    <NoQuestions : Integer><NoAnswers : Integer><NoAuthRecords : Integer>
    <NoAdditionalRecords : Integer><DomainName : FDQN><Type : Integer>
    <Class : Integer><TTL : Integer><RDLENGTH : Integer>
    <RDATA : String [ f−RDATA( Type ) ]>
<EndTypes>

<Header : DNS>
    <ID:16><QR:1><Opcode:4><AA:1><TC:1><RD:1><RA:1><Z:3><Rcode:4>
    <NoQuestions:16><NoAnswers:16><NoAuthRecords:16><NoAdditionalRecords:16>
<End : Header>

<Message : DNSQuestion>
    <Rule :QR=false>
    <DomainName:\0><Type:16><Class:16>
<End : Message>

<Message : DNSResponse>
    <Rule :QR=true>
    <DomainName:\0><Type:16><Class:16>
    <TTL:32><RDLENGTH:16><RDATA : RDLENGTH>
<End : Message>
```

Figure 4.11: Bonjour mDNS Message Description

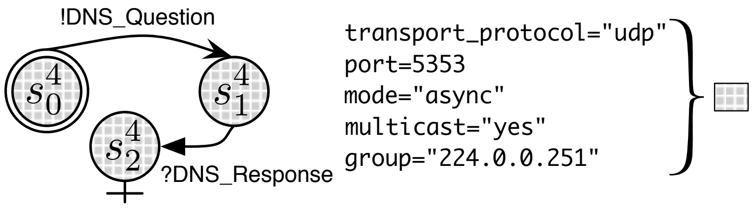


Figure 4.12: mDNS coloured automaton

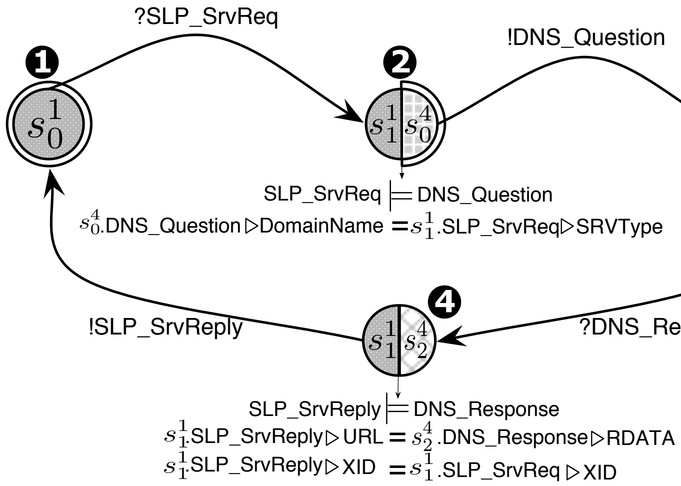


Figure 4.13: A merged automaton for SLP and mDNS protocols

client was executed, Starlink executed the merged automaton and successfully sent an SLP `SrvReply` message composed of the content from the SSDP and HTTP fields.

```
<Types>
<Method : String >
      <URI : String >
      < Version : String >
      <MX: Integer >
      <MM: Integer >
      <MAN: String >
      <USN: String >
       . . .
<EndTypes>

<Header : SSDP>
      <Method:32 >
      <URI:32 >
      < Version:13,10 >
      < Fields :13 ,10:58 >
<End : Header>

<Message : SSDP_Search>
      <Rule : Method=M-SEARCH>
<End : Message>

<Message : SSDP_Response>
      <Rule : Method=HTTP/1.1 >
<End : Message>

<Message : SSDP_Notify>
      <Rule : Method=NOTIFY>
<End : Message>
```

Figure 4.14: Partial SSDP Message Description

4.4.3 Evaluation

Originally we stated that we required the following from CONNECT:

- We require transparent interoperability. In the case study, the legacy protocols are implemented and deployed independently, they are never aware of the framework and hence the case studies show that transparent interoperability has been achieved.
- Offer rich translations. The case studies show that we can translate correctly between three different protocols that are heterogeneous in terms of protocol sequences, e.g. SLP compared to UPnP, and heterogeneous in terms of message content, e.g., binary to text messages.
- Minimise development effort. In this case study we only needed to provide high-level models, there was no low-level programming. Further, we were able to reuse the models across the cases, i.e., we only needed to model SLP once and then write only the merged LTS for the particular case (typically, these LTS are around 100 lines of XML, but this depends on the complexity of the translation).

The performance of the CONNECTors were evaluated by investigating the time taken to perform interoperability translation. We then compared this to the typical responsiveness of discovery protocols in terms of the time taken to return a service reply to a lookup request. Figure 4.15(a) shows the results of the bench measures of the individual protocols (these are measures of the legacy applications implemented using OpenSLP for SLP, the Apple Windows SDK for Bonjour and Cyberlink for UPnP). To obtain the measures, we calculated the time from when the client sent the message until the response was received. For each case, we repeated the experiment 100 times and took the min, max, median of these results.

All experiments were performed with the client and the service on the same machine (3 Ghz CPU, 2Gb memory running Windows Vista Operating System, the Java VM was version 1.6.2) to avoid measuring additional network latency, which may not be constant.

Subsequently, we measured the time taken to translate from one protocol to another within a CONNECTor. This measured the time from when the message was first received by the framework until the

Response time measures for legacy discovery protocols

Protocol	Min (ms)	Median (ms)	Max (ms)
SLP	5982	6022	6053
Bonjour	687	710	726
UPnP	945	1014	1079

(a)

Translation times of Starlink connectors

Case	Min(ms)	Median (ms)	Max (ms)
1. SLP to UPnP	319	337	343
2. SLP to Bonjour	255	271	287
3. UPnP to SLP	6208	6311	6450
4. UPnP to Bonjour	253	289	311
5. Bonjour to UPnP	334	359	379
6. Bonjour to SLP	6168	6190	6244

(b)

Figure 4.15: Native service discovery vs. CONNECTor (ms)

translated output response was sent on the output socket. Figure 4.15(b) shows these measures. We can see from the results that there is a significant but varied expense to additional translation: in case 6 it is approximately a 600 percentage increase in response time, while in case 1 it is 5 percent. This is because the cost of translation is bounded by the response of the legacy protocols; if SLP takes 6 seconds to respond that is added to the translation. However, in the domain of service discovery protocols the timeout of the request response is generally in terms of seconds (OpenSLP sets the default timeout to 15 seconds, while Cyberlink does not bound the response time); all of the results are within this range and the solution is both possible and acceptable.

4.5 Handcrafting CONNECTors: Universal Instant Messenger

4.5.1 Goals

The goal of this experiment is to hand-code a mediator achieving transparent interoperability between various instant messaging (IM) applications, namely Windows Live Messenger[4](previously called MSN messenger), Yahoo! Messenger[5], and Google Talk[6], as an initial step to gauge the complexity of the synthesis and to study the CONNECT approach to interoperability focusing on mediation at the application and messaging layers.

The IM applications under consideration offer similar functionalities such as managing a list of contacts or exchanging textual messages. However, a user of Yahoo! Messenger is unable to exchange instant messages with a user of Google Talk. Indeed, there is no common standard for IM, so users have to maintain multiple accounts in order to interact with each other. The situation does not make any sense from the end-user perspective, but unfortunately it reflects the way IM (like many other existing applications) has developed. Hence, solutions that tackle the problem of disparate applications and enable transparent interaction across different communities are needed.

Microsoft and Yahoo developed a bridging solution in order to make Windows Live Messenger and Yahoo! Messenger interoperable [46]. However, bridging is infeasible in the long term as the number of IM systems grows continually. Enterprise Service Buses (ESBs) offers an alternative by providing an intermediary message bus to allow N-1-M mappings between system messages. Examples of ESB that ensure interoperability between IM systems are Apache Synapse IM Mediator[7] and WSO2 ESB[8], both of

[4] http://explore.live.com/windows-live-messenger/
[5] http://messenger.yahoo.com/
[6] http://www.google.com/talk/
[7] http://esbsite.org/
[8] http://wso2.com/products/enterprise-service-bus/

which use the Extensible Messaging and Presence Protocol[9] (XMPP) as a standard intermediary protocol.

Another solution consists in using a single application providing a uniform interface to interact with several instant messaging systems. The application performs the translation between the uniform interface and the existing IM protocols. However, it requires the user to install new software and to have many identities belonging to multiple IM providers. Moreover, these solutions, such as Pidgin[10] or Adium[11], generally provide a subset of the features provided by the original proprietary applications.

Finally, transparent interoperability solutions translate protocol-specific messages, behavior and data to and from an intermediary representation. An example of such an approach is CrossTalk [50] which uses XMPP as a standard intermediate protocol. However, relying on a fixed intermediary protocol or specification might become restrictive over time. Ontologies provide an extensible way to define this common specification.

To sum up, this experiment addresses the following CONNECT interoperability dimensions:

- *Data heterogeneity*. MSN Messenger protocol (MSNP), the protocol used for Windows Live Messenger, uses text-based messages whose structure includes several constants with predefined values. On the other hand, Yahoo! Messenger Protocol (YMSG) defines binary messages that include a header and key-value pairs. Finally, the XMPP messages are defined in terms of XML Schema.

- *Application heterogeneity*. The CONNECT networked system model should include the behavior of the networked systems, either specified by the developer or learned using the learning enabler. However, both MSNP and YMSG are proprietary protocols and do not provide any specification of their behavior. Therefore, we reverse engineered these proprietary protocols to compare them with the ones learned automatically. Also, even though the protocols are simple and quite similar each one communicates with its own proprietary application server used for authentication and for relying the messages between instant messaging application. Hence, each protocol defines how to establish a session in a slightly different manner.

- *Heterogeneity of Non-functional properties*. MSNP and YMSG do not encrypt their messages whereas Google Talk activates the encryption option over XMPP making it difficult to access and modify the content of the messages transparently.

4.5.2 Methodology

Understanding the protocols

One of the challenges we had to face is that neither MSNP nor YMSG have a specification of their protocols. We had then to reverse engineer the protocol using: (i) network logs collected by monitoring the protocols using Wireshark[12], (ii) client-side APIs, and (iii) unofficial online documentation. On the other hand, Google Talk is based on XMPP, which is an IETF standard and hence has a detailed specification. Figures 4.16, 4.17, and 4.18 illustrate the behaviour of the MSNP, YMSG and Google Talk protocols respectively. Note that for the sake of readability, we only point out the relevant inputs and outputs abstracting away the details used to generate the messages such as constants and packet length, we also use a box to denote the transitions in parallel.

Demonstrating CONNECT solution to application-layer interoperability

The IM applications we considered support the configuration of a proxy server which we used for connecting them to the mediator using SOCKS[13] protocol. The proxy, `SOCKSProxy`, has been implemented using JSOCKS API[14] (see Figure 4.19).

The `UserConnection` module manages the user connection and its related data. The `BuddyManagement` module maintains the list of contacts of each user.

[9] http://www.xmpp.org/
[10] http://www.pidgin.im/
[11] http://adium.im/
[12] http://www.wireshark.org/
[13] http://tools.ietf.org/html/rfc1928
[14] http://jsocks.sourceforge.net/

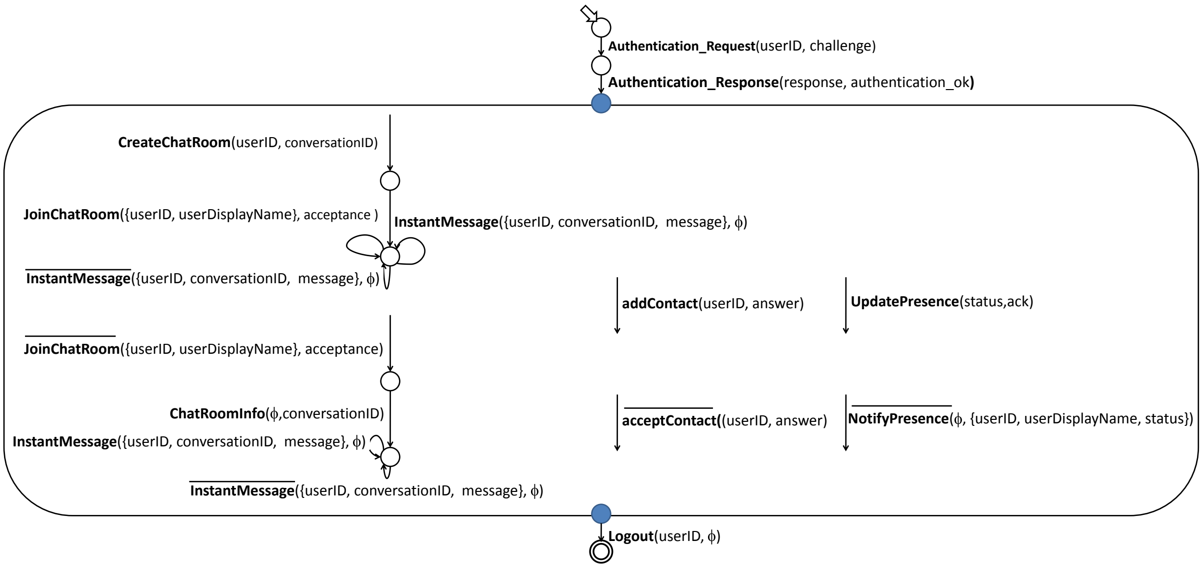


Figure 4.16: MSNP protocol description

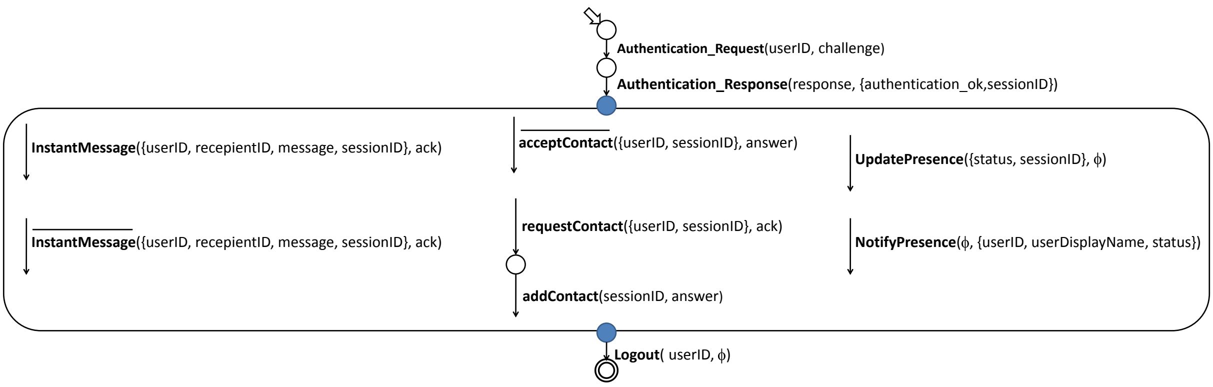


Figure 4.17: YMSG protocol description

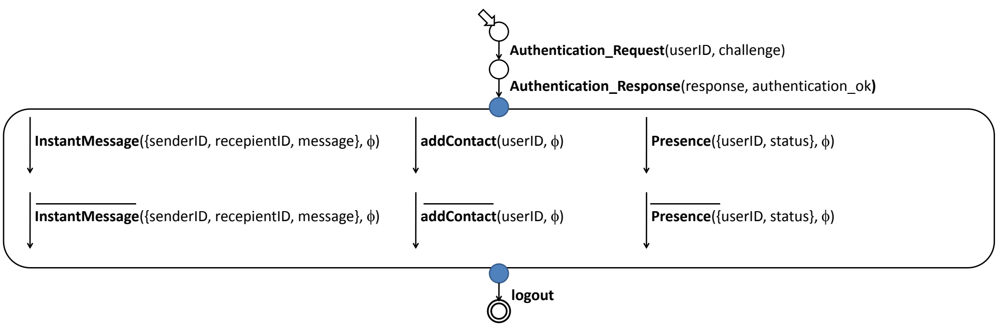


Figure 4.18: Google talk protocol description

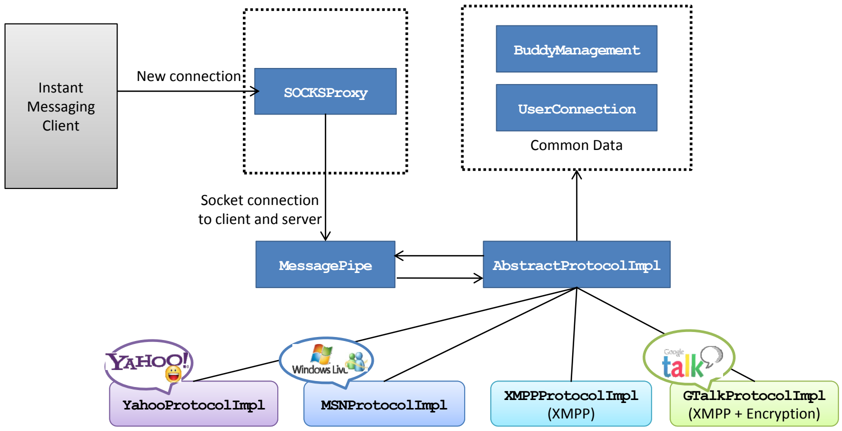


Figure 4.19: Universal Instant Messaging: the overall architecture

The `AbstractProtocolImpl` interface defines the functions required to support any IM protocol. An implementation of this interface (`YahooProtocolImp`, `MSNProtocolImp`, and `GTalkProtocolImp`) is associated with each protocol (YMSG, MSNP, and Google Talk respectively) to specify how the specific messages are parsed and composed to handle the behavior of each protocol.

There are a number of mismatches between protocols we have to cope with:

- *Typing Notifications in MSNP*. In MSNP, a notification is sent as soon as the user starts/ends typing, a message can only be received if a notification has been sent beforehand. Since other IM protocols do not require a typing notification before sending a message, the mediator generates a typing notification and sends it to the Windows Live Messenger application before sending a message. This action corresponds to the message producer pattern defined in D3.2 [18].

- *Session identifier in YMSG*. A YMSG packet header contains a unique `sessionID` provided by the Yahoo server during authentication and required for all following messages. The mediator stores the `sessionID` and embeds it in all the messages.

- *Message acknowledgment in YMSG*. In YMSG, after sending a message (identified by a `messageID`), an acknowledgment with the same `messageID` is expected, otherwise the message is sent again. The mediator sends the appropriate acknowledgment for the user.

- *Adding a contact in YMSG*. To add a new contact using YMSG, a message is first sent to the server, which checks that the new contact has an appropriate email address, that is an email address belonging to the *Yahoo* or *MSN* domains. Only after the email address has been approved by the server, can it be added to the user contact list. However, since we would like to allow users to communicate whatever IM application or address they use, the mediator intercepts the add-contact request and sends a positive response to the Yahoo! Messenger application in order to add the new contact.

Dealing with encryption

Unlike Windows Live Messenger and Yahoo! Messenger, Google Talk enforces encryption over the XMPP messages. Consequently, the mediator cannot parse or modify these messages. To overcome this limitation, the mediator makes use of a Google Talk robot (bot) (see Figure 4.20), added by the user in order to include commands in the messages. These messages are: `Help` to get the list of available commands, `IM <destinationID> <message>` to send a message `message` to user `destinationID`, or `AVAILUSERS` to get the list of online users.

The bot is implemented using the Smack API[15] and is integrated as any other user with an implementation of the `AbstractProtocolImpl` interface. Even if this solution is less transparent than the

[15] http://www.igniterealtime.org/projects/smack/

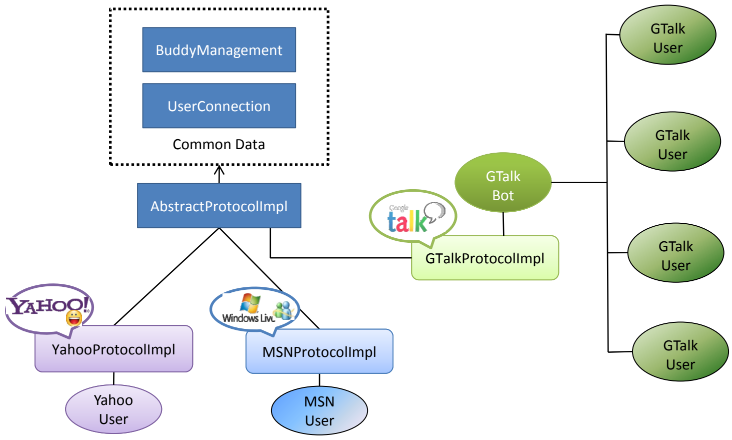


Figure 4.20: Dealing with encryption in Universal Instant Messaging

previous one since it forces the user to add a bot to its contact list and to use extra commands, it remains the only viable one to bypass the encryption process.

4.5.3 Evaluation

As an initial step, we developed a mediator between heterogeneous instant messaging (IM) applications. The development of this mediator allowed us to estimate the complexity of its generation, as well as the limits of transparency in case of encryption. We are currently automating the generation of the mediator through the use of CONNECT semantic-based techniques in order to automatically check the compatibility of the different IM protocols and to synthesize the mediator that makes them communicate. Therefore, our ongoing work relates to:

- defining an ontology of instant messaging systems,
- using the IM ontology to automatically check the behavioral compatibility of the different protocols based on behavioral matchmaking described in Section 3.3.2, and
- designing and implementing a solution to automatically map the messages of different IM applications based on their ontological representation. Toward this end, we use listeners/actuators, described in Section 4.2, to perform the translation between concrete and abstract messages and use XSL transformations[16] to achieve the mapping between the abstract messages and the corresponding ontological concepts.

Furthermore, this prototype will serve as a reference implementation to compare the automated solutions to the hard-coded one, i.e., the behavior automatically learned with the one obtained by reverse-engineering the protocols, and the mediator automatically synthesized with the hard-coded one.

4.6 Conclusions

Here, we have highlighted that significant progress has been made towards achieving software prototypes that can deploy CONNECTors which will ensure highly heterogeneous systems and protocols can interoperate with one another. These software prototypes are currently:

- Java implemented, generic and customisable Listeners and Actuators. These can interpret high-level message format spefications for composition and parsing of legacy protocol information. Binary and text versions of the Listeners and Actuators have been developed.

[16] http://www.w3.org/TR/xslt

- A Java implemented mediator engine for k-coloured LTS. This can execute a generated LTS that describes the translation of one protocol to one or more other protocols. This prototype can equally be used for application and middleware protocol translation.
- A Java implemented mediator that solves data and behavioral heterogeneities between IM applications. The aim is to evaluate the effort to generate the mediator using CONNECT principles to interoperability.

Further, we have identified the potential of high-level models (as advocated across the CONNECT project) to also handle the often complex interaction with diverse legacy communication protocols. That is, the very low-level details that are often a barrier to interoperability can be addressed by modelling and generation of a solution. Following such a philosophy ensures that the overall CONNECT approach and prototypes will remain future proof.

Finally, we have investigated the realisation of CONNECTors to join multiple protocols; this work has verified (using two case studies) that the designed CONNECTor architecture is fit for purpose and achieves the required interoperability. These experiments consider interoperability at the application level (in the IM example) and at the middleware protocol level (in the discovery example). It is a key objective for the third year to consider these in tandem and produce CONNECTors for heterogenous applications deployed on heterogeneous middleware.

5 CONNECTor Deployment

5.1 Introduction

The objective of this chapter is to describe the operation of the deployment enabler; here we present how the artifacts that result from the synthesis process (i.e., after the concrete CONNECTors have been realised) are deployed and started up. As explained previously (See Section 2.4, Figure 2.4 page 23), CONNECTors are made of two parts: a mediator, whose code is generated from the concrete CONNECTor model that results from the synthesis process (as documented in Deliverable D3.2 [18]; and the Listener and Actuator classes that are generated from models of middleware protocols. Importantly, these Listener and Actuator classes can be re-used from one CONNECTor to another, and hence the deployment enabler maintains an *Interoperability Library* from which these can be drawn.

The first part of this chapter outlines the overall deployment process for a CONNECTor; this describes how the procedure is initiated by the synthesis of the required concrete CONNECTor model between two networked systems, and then illustrates how the running CONNECTor artifacts are deployed in the network environment and then executed. Section 5.3 focuses on the packaging and deployment of the Listeners and Actuators that provide the proxies needed to properly communicate with versatile middleware technologies. Section 5.4 focuses on the deployment and starting up of the mediator itself, and explains how we leverage existing services provided by the OSGi platform to properly resolve the dependencies between the mediator and the required interoperability proxies. The last section then introduces an alternative synthesis/deployment solution, based on service orchestrations, which will be investigated in the last year of the project.

5.2 Overview of the Deployment Process

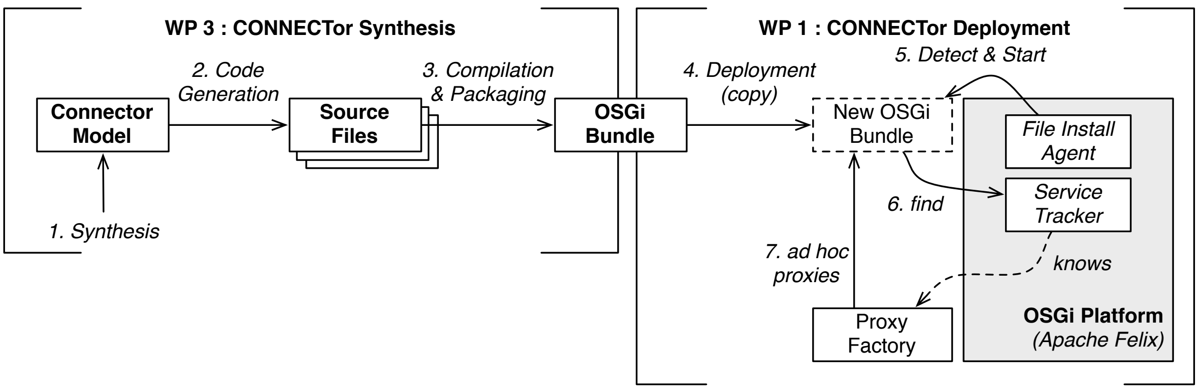


Figure 5.1: Overview of the CONNECTor deployment process

Figure 5.1, above, shows the overall deployment process, starting from the synthesis of the CONNECTor model and continuing to the final deployment of the resulting OSGi bundles. This process is composed of the following steps:

1. The initial step is the synthesis of the CONNECTor model (step 1 in Figure 5.1), which is obtained after the analysis of the characteristics of the networked systems at play in the collaboration. This model, which only describes the expected behavior of the mediator, is detailed in the Deliverable D3.2 [18].

2. The CONNECTor model is then compiled into a set of source files (including: several Java classes, an Ant build file and a manifest file). The code generation has been outlined in Section 4.3.1 and is further detailed in the Deliverable D3.2 [18].

3. The resulting source files are then compiled and packaged into a self-described Jar file (step 3 in Figure 5.1), compliant with OSGi standards, that can be consequently deployed in any OSGi execution platform.
4. The target platform is the Apache Felix OSGi platform: it includes a *File Install Agent* that is able to detect and start new OSGi bundles that are placed into predefined directories (step 5 in Figure 5.1). This boils the deployment down to a file transfer of the CONNECTor binaries into the correct directory on the target host, i.e., the physical machine that the CONNECTor will operate upon. (step 4 in Figure 5.1).
5. When starting up, the mediator leverages the service discovery feature provided by the OSGi platform to locate the needed proxy factories. While deploying a new CONNECTor, we indeed assume that the interoperability libraries, so called *Proxy Factories*, providing Listeners and Actuators have been previously deployed and registered on the target host. If the required Listeners and Actuators are not deployed then these can be obtained from the deployment enabler's Interoperability library.
6. Finally the mediator uses these factories to create a separated thread for each partner involved in the collaboration (step 7 in Figure 5.1). Each thread is waiting for messages coming from a given partner and the mediator is then ready to proceed.

5.3 Building Proxy Factories

As explained in previous chapters, a running CONNECTor is made of two parts: a mediator part and several interoperability libraries (listeners and actuators). The former depends on the application and is generated from the CONNECTor models, whereas the latter ones depend on the middleware protocols employed by the application and, due to the nature of middleware protocols, can consequently be reused from one CONNECTor to another. As a result, these two parts are separately deployed on the target OSGi platform. This section focuses on the implementation and on the deployment of the *proxy factory*, which appear in Figure 5.1.

For the record, messages can be read/written according to a description of the corresponding middleware protocol (See Section 4.2). Such protocol descriptions are actually used to generate a parser object and a composer object that will decode and encode messages respectively. The proxy factory is thus responsible for the creation of an object, so-called *proxy*, in charge of the connection with a given partner (on a given port), and which ensures the exchange of messages with this partner.

The proxy object is a separated thread which is in charge of listening for incoming messages. The following code excerpt illustrates the main behavior of this thread. While the thread has not been stopped, the proxy keeps reading data on the open connection (lines 5 to 9). It's worth to note that the *readLine* method invoked on line 5 is blocking, avoiding an active wait for data. Once a message has been read, the proxy will use a specific parser object to build an *Abstract Message* (see line 11) and will then pass this to the mediator object (line 12).

```
1  public void run() {
2
3    while(!stopped) {
4      try {
5        String line = reader.readLine();
6        while(reader.ready()){
7          buffer.append(line);
8          line = reader.readLine();
9        }
10
11       Message message = parser.parse(buffer.getByte());
12       mediator.process(message);
13
14       buffer.setLength(0);
15     } catch (IOException e) {
16       // ...
17     }
18   }
19 }
```

The proxy may also receive some outgoing messages from the mediator. In this situation, the proxy object uses a specific composer object in order to transform the Abstract Message into an array of bytes that can be communicated via the network transport connection with the partner. The following code illustrates this simple data manipulation.

```java
public void send(Message message) {
  byte[] data = this.composer.composer(Message);
  connection.outputStream.write(data);
  connection.outputStream.flush();
}
```

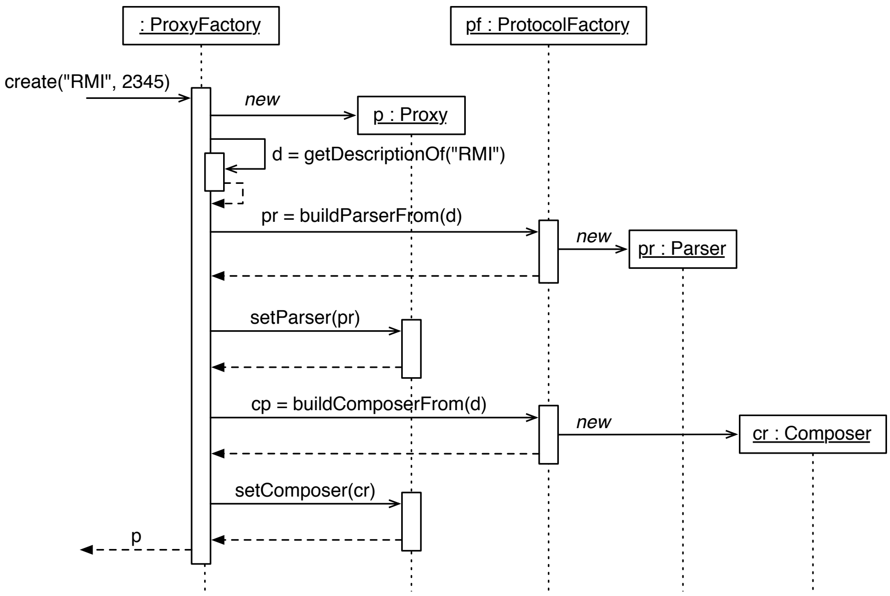


Figure 5.2: Proxy Factory Service: On-demand construction of proxy

The construction of a specific proxy object is illustrated on Figure 5.2. The proxy factory can be invoked to create a proxy object that will listen for messages of a given protocol, on a given port. First the proxy factory creates a new proxy object. The proxy factory then retrieves from its local database the description of the protocol that is demanded and asks the *ProtocolFactory* object to create the relevant parser and composer objects. These two objects are then used to configure the proxy previously initialized. We assume that the descriptions of all existing protocols that are supported by the discovery enabler are registered in the *ProxyFactory* object. It worth to note that the proxy object is also configured with the calling mediator object, and the port on which it must open a connection, but this is not depicted in Figure 5.2 for the sake clarity.

The proxy factory is deployed as an OSGi service, which can be accessed using the OSGi service tracker. The following code excerpt illustrates how the mediator object obtains a relevant reference on the *ProxyFactory* in order to generate the proxy that it needs.

```java
public void start(BundleContext bundleContext) throws Exception {
  Activator.context = bundleContext;
  ServiceReference proxyFactoryRef = context.getServiceReference("eu.connect.osgi.ProxyFactory")
      ;
  ProxyFactory proxyfactory = (ProxyFactory) context.getService(proxyFactoryRef);
  Proxy rmiProxy = proxyFactory.create("RMI", 2345, this.mediator);
}
```

5.4 Deployment of a Compiled Mediator

As mentioned in the overview of the deployment process (see Figure 5.1), the deployment of the mediator bundle is boiled down to the copy of the associated Jar file into a predefined directory. The *File Install Agent* detects the new bundles and starts them automatically. While the mediator is starting, it uses the *Service Tracker* to access the Proxy Factory Service and dynamically builds the proxy objects that are needed. Once the mediator is ready, its architecture will look like the UML collaboration diagram presented Figure 5.3 below.

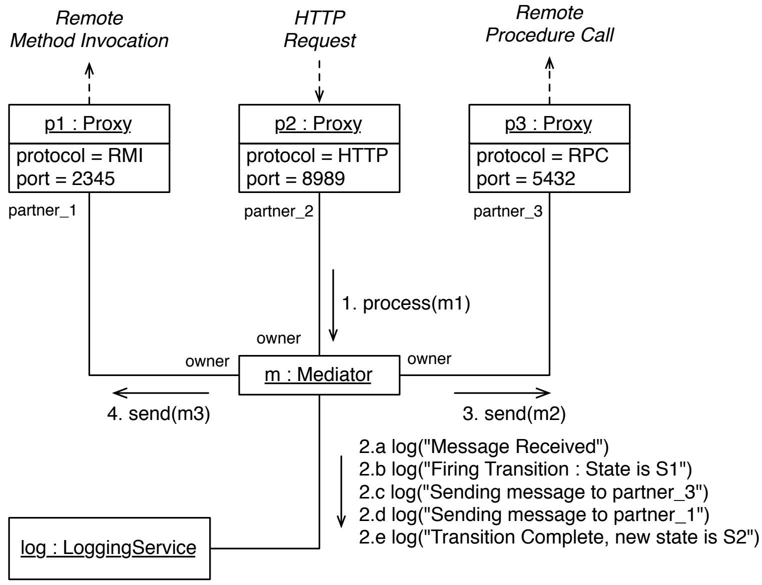


Figure 5.3: Architecture of a deployed CONNECTor mediating between three partners using three different communication protocols

Figure 5.3 illustrates the result of the deployment: it shows the interactions between the mediator part and the three proxy objects to three different partners using RMI, HTTP and RPC respectively. When a HTTP request is received by the proxy 2 (p2), it parses the request and transmits the corresponding message to the mediator object. The mediator first stores this message, and later processes it. The message will trigger one of the transitions of the underlying automaton, and the mediator builds and sends two messages to the proxies 3 and 1 respectively. As a result, the two proxies output on the network the proper invocations. All the actions performed by the mediator while processing a message are recorded using the OSGi standard logging service.

The architecture of the runtime CONNECTor implies that each CONNECTor is composed of at least 3 threads: one for the mediator, plus one for each partner (there are at least two). A large number of threads may significantly slow down the underlying Java virtual machine and consequently raise performance issues if many connectors are deployed on the same host. An alternative design such as the Reactor pattern will be investigated during the third year in order to reduce, or at best limit, the number of threads needed to implement a CONNECTor.

5.5 Towards the Deployment of Model-based Mediators

We have shown here the details of the software prototype and tools that compose the behaviour of the deployment enabler; this is successfully able to deploy the code-generation based CONNECTors that were described both in the previous section 4 and the Deliverabler D3.2 [18]. However, the project is also investigating different strategies to realise CONNECTors. As described in Section 4.3, model interpretation is an alternative solution to the generation of ad-hoc CONNECTors whose code is generated, compiled, and later deployed on a remote host. The two solutions do not conflict, rather they are suited to different situations. The use of code generation for *ad hoc* CONNECTors ensures good performances but provides a low degree of maintainability. This fits perfectly the cases where partners are legacy systems that are not likely to change. By contrast, the dynamic interpretation of the CONNECTor models, which provides lower performance but much better flexibility, better fits systems where partners are likely to dynamically appear or disappear.

As mentioned in Section 4.3.2 we have started to investigate the generation of BPEL code from the CONNECTor models. Because there are various similarities between a CONNECTor and a service orchestration, we plan to leverage existing orchestration engines, such as Apache ODE for instance to dynamically execute the CONNECTor. However, CONNECTors go further than service orchestrations, which implies the assumption that each partner involved in the collaboration is indeed a web service, exchanging SOAP messages. There is consequently a need for an extension of the existing orchestration engines, in order to support middleware technologies on demand, as we did for the ad hoc CONNECTors. Additional details about the extension of the Apache ODE can be found in the Deliverable D3.2 [18].

With respect to k-coloured LTS, which are implemented solely in Java, we are investigating the same deployment approach used for the code-generated mediators, i.e., using OSGi-based deployments. Given that k-coloured LTS are self-contained executables that can be packaged in JAR files, we believe that the OSGi tools and approaches are similarly well suited.

5.6 Conclusion

In conclusion, we have presented a detailed description of deployment enabler; we have shown how CONNECT deploys CONNECTors in the networked environment in order to allow the legacy networked systems to communicate with one another. Naturally, this work has concentrated on the output of the synthesis enabler at this stage in the project, i.e., it considers the software mediators that have been generated from high-level models. However, other mediator implementations have been investigated within the project, which execute the behaviour prescribed by a model of the mediator. Future work in the area of the deployment enabler will concentrate on these further (it is already well placed to handle these methods) in the third year of the project.

6 The Role of Ontologies

6.1 Introduction

In this chapter we discuss the role of ontologies in addressing the interoperability problem, with the goal to contribute to the overall construction of a CONNECTor. The rational for looking at ontologies is that they provide the capability to reason about a given phenomena, or a given system, or, as in the case of CONNECT, on a specific interoperability problem. The ability to reason about information which is afforded by ontologies, provides us with an automatic way to identify the conditions under which different middlewares can interoperate. For example, reasoning about the middlewares may reveal the different ways of doing discovery in different systems so that the CONNECTor will find a way to bridge between the two systems. In this section, we will explore the use of ontologies to address interoperability of different middlewares, addressing the problem at different levels, from as low as the messaging level, to the application level. Such ideas go beyond the state of the art in the application of semantic technologies, which has typically concentrated on semantic web technologies.

The rest of this section is organized as follows. In section 6.2 we provide a definition of ontology and its use; then we discuss two examples of use of ontologies, the first one, in section 6.3, proposes a model of messages in the VANET context, the second one, in section 6.4, shows the use ontologies to reason about systems architecture. In these two examples, we tackle very different problems at different levels of abstraction, specifically we look at (1) mapping packet structures and (2) reasoning about architecture patterns, but we will converge to a similar approach and solution. Finally, in section 6.5 we synthesize the use of ontologies in CONNECT at large, and we try to extract what are the main advantages and problems related with their use.

6.2 Definition of Ontology

The universally accepted definition of ontology in Computer Science was proposed by Tom Gruber, who formulated it as the following: *An ontology is a specification of a conceptualization.*[1]. Gruber provided also a longer and somewhat more sophisticated definition of Ontologiy in the *Encyclopedia of Database Systems*[2], where he expands on the initial definition without affecting its meaning drastically.

In the context of CONNECT Gruber's definition is hardly satisfactory. First of all, any data structure is a conceptualization, and therefore it is not quite clear what is the reason for using ontologies in the context of this project. Second, and more importantly, Gruber's definition does not highlight the computational mechanisms that underly the use of ontologies. As a consequence, this definition hides how the use of ontologies affects interoperability.

In our work in CONNECT, we adopt a different approach where an ontology is a tuple$<$A,L,P$>$ where A is a set of axioms, which implements the conceptualization highlighted in Gruber's definition, L is a language in which to express these axioms, and P is a proof theory, that supports the automatic derivation of consequences from the axioms. In the rest of the discussion, we use the word *logic* to mean the combination of language L and proof theory P. Defining ontology in terms of its logics, allows us to uncover the computational mechanisms underlying the ontology and therefore investigate how its use affects the the interoperability of the whole system.

Indeed, the definition above raises two issues. the first one is, what logic is adopted to represent ontologies? The second one is, what interoperability issues are raised by the use of ontologies? In the next two subsections we will look at these two issues.

6.2.1 Logics for Ontologies

In principle there is a wealth of logics that have been implemented to support the reasonings about data but the emergence of a set of standards around the Semantic Web initiative[3] at W3C greatly limits the

[1] http://www-ksl.stanford.edu/kst/what-is-an-ontology.html
[2] http://tomgruber.org/writing/ontology-definition-2007.htm
[3] http://www.w3.org/2001/sw

scope of the decision. The flagship of these standards is OWL, a family of languages defined around the RDF[4] serialization of Description Logics [2].

Our work shows that OWL is not sufficient to express the concepts that we want to express in CONNECT. As a consequence, we adopt also SWRL[5] and RIF[6], which are rule extensions of OWL, and SQWRL[7] and SPARQL[8], which allow to query the knowledge base. In the next sections we explore these languages. The purpose of our exploration is not to provide a tutorial, but rather to highlight features that we use, and problems that we encountered during our work.

OWL

OWL, which stands for *Ontology Web Language*, is an ontology language that belongs to a family of logics that stands under the label of Description Logics (hereafter DL)[9]. Within OWL it is possible to express two types of descriptions: *terminological* descriptions, or TBOX, that specify *Classes* that define the important types of objects in the domain, and *Properties* that define relations between classes; and *assertion* descriptions, or ABOX, that describe the model of actual objects in the domain. For example, if we define an ontology of computational systems, we may want to distinguish between *Client* and *Server*, a fragment of such an ontology is displayed in figure 6.8, as well as properties such as *hasClient* or *hasServer* that specify the relation between two different systems. All these statements define the TBOX of the ontology. Using this TBOX, it is then possible to describe instances of actual computational mechanisms such as *myBrowser* as a client, and *google* as a server. These descriptions are stored in the ontology ABOX.

Superficially, the structure of an OWL ontology seems to mimic the Object Oriented structure of classes, subclasses, and instances. The advantage of defining classes in OWL is that one can take advantage of the OWL inferencing. Whereas in OOP a programmer is forced to specify the type of every instance that he creates, in OWL this is not required. Indeed, the programmer may just define *myBrowser* as an instance of *Thing*, the root of the class taxonomy, and all the properties of the object. Upon running the inference engine, *myBrowser* is classified as of type client. In addition to classification, OWL provides a second type of inference named *class subsumption* which identifies if all conditions are satisfied to state that a there is a subclass relation between two classes. Again, on the opposite of OOP, the subclass relation between two classes should not be specified statically, but rather it can be inferred on the bases of the statements that have been made about these classes. Continuing with the example above, I may not specify explicitly that the class *WebClient* is a subclass of *Client* nor that *WebServer* is a subclass of *Server*, but by specifying constraints among them the subclass relation will be inferred directly by the inference engine.

It is worth noticing that, because of the ability of classifing instances as well as computing class subsumption, it is very limiting to think of OWL as a pure taxonomic language. Rather the taxonomy emerges through the relations that have been specified across the different concepts. Indeed, OWL classes are better thought as sets of logic assertions. To this extent OWL provides a reach language to make assertions, a programmer could for instance express cardinality constraints such as "*the car has at most 6 wheels*" or that "*a horse has at exactly 4 legs*" as well as universal statements which allow to specify concepts like "*a WebClient is a client whose servers are all WebServers*", as well as existential statements such as "*a WebServer has some clients that are browsers*", which may be true when some of the clients are not browsers.

As pointed out above, OWL is a family of logics rather than a single logic. The initial specification of OWL defined three logics: OWL Light, OWL DL, and OWL Full, with different levels of espressivity and computation complexity. For example, in OWL Light it was possible to specify a concept like "*a car has more than one wheel*" but not that "*a car has exactly 4 wheels*", while the same concept could be expressed in OWL DL; furthermore in OWL Full it is possible to relate instances to classes, and therefore it was possible to express a concept like "*a bookstore sells things of type Book*" which relates an instance, *my bookstore*, with a class *Book*. This concept of bookshop could not be specified in the other two logics.

[4] http://www.w3.org/rdf

[5] http://www.w3.org/Submission/SWRL/

[6] http://www.w3.org/2005/rules/wiki/RIF_Working_Group

[7] http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL

[8] http://www.w3.org/TR/rdf-sparql-query/

[9] An extensive description of the different statements that can be expressed in DL logics, as well as OWL, is presented in http://en.wikipedia.org/wiki/Descriptionlogic

The situation now is even more complex since with the introduction of OWL 2; additional constraints can be expressed such as property chains and arithmetic values. Therefore in OWL 2 it is possible to express concepts like "*X is the uncle of Y if X is the brother of Z and Z is the a parent of Y*", furthermore it is possible to assert that "*X is adult if aged more than 18*". Neither of these concepts could be expressed in OWL 1.

Despite the different expressivity of the different OWL species, if OWL Full is excluded, they are contained one in the other, therefore, in principle, the different degrees of expressivity have no consequence on interoperability. If System 1 refers to an ontology in OWL Light, and System 2 to an ontology in OWL 2, any reasoner for OWL2 will also be able to handle OWL Light. The converse is of course not true, therefore minimally the use of OWL imposes some requirements on the computational infrastructure of the CONNECTor constructor. But in general there may be a number of different reasons for systems to limit themselves to OWL Light; among the many reasons, one may be efficiency: the computational complexity, of OWL Light is much lower than OWL 2, and typically the performance is bound to be better.

SWRL and RIF

The problem with using OWL is that even OWL 2 does not provide all the expressivity that is required to model computational systems. Specifically, for computational reasons the models supported by OWL and in general by DL logics are tree structures. In turn, there is no way for an instance to refer to the value of other instances. We encountered this problem in the modeling of the demonstration described in the Deliverable D6.2 and highlighted this below in Section 6.4. There is would be desirable to state when two systems are in the same domain. But in OWL it is impossible to state a concept like "*Syst1.domain=Syst2.domain*", along the same line we cannot express constraints such as "*the total time lapse is the sum of all the individual time lapses*".

SWRL, Semantic Web Rule Language, provides a way to go beyond these problems, by defining a rule language on top of OWL which allows one to overcome some of the restrictions that are present in OWL. Essentially, SWRL allows us to specify path properties such as the following: "$parent(?x, ?y) \wedge brother(?y, ?z) \Rightarrow uncle(?x, ?z)$" which were not possible in OWL 1, and it adds to the language a number of operators such as math operators, string manipulation and equality that were not possible in OWL 1, and are marginally available in OWL2. Using SWRL, we can address the problem of constraining the systems domains above by defining a rule like the following "$domain(?s1, ?d) \wedge domain(?s2, ?d) \Rightarrow samedomain(?s1, ?s2)$", where ?s1 and ?s2 are two variables to be bound to the two systems, and ?d is a variable bound to a domain; the rule holds only when ?d has the same value in both domains. The cost of using SWRL in conjunction with OWL is that it can be proven that the logics resulting from SWRL and OWL is undecidable. In our opinion, the loss of decidability is surely compensated by the gain in expressivity.

SWRL never reached the standard level at W3C, being instead defined as member submission, which highlights technologies that are of importance for the Web, and may require further standardization work. In the case of SWRL, the standardization work has been taken up by the Rule Interchange Format (RIF) effort, which is defining a rule language on top of the semantic web languages. The RIF effort is still on-going, but no viable tools are available at this time. So for the time being we are limiting ourself to SWRL.

SQWRL and SPARQL

SQWRL and SPARQL are query language that allow to extract information out of the knowledge base. Essentially, they view the knowledge base as a graph structure and extract from the graph those nodes that correspond to a given pattern. A simple example[10] of a SPARQL query is given below. The result of such a query is to extract the url of "Jon Foobar" blog.

```
SELECT ?url
FROM  <bloggers.rdf>
WHERE  {
    ?contributor foaf:name "Jon Foobar" .
```

[10] The example below is taken from the introduction to SPARQL in http://www.ibm.com/developerworks/xml/library/j-sparql/

```
    ?contributor foaf:weblog ?url .
}
```

Query languages are important in CONNECT because they provide a way to reason about the different properties that define the objects in the Knowledge Base, and in particular to find which properties belong to one object but not to the other. In turn this information provides the bases to reason about what transformations are required to merge two different objects. A very clear example of this difference is provided in section 6.3.4 below.

While SQWRL and SPARQL are both query languages, they are very different in the details. But, from the point of view of the CONNECT project though, the difference is mostly based on a pragmatic set of decision. SQWRL is supported by and earlier version of the Protege tool, that is the main tool for ontology development, and it is very well integrated with SWRL. For this reason the use of SQWRL provides a powerful way to interact with ontologies. On the other side SPARQL reached the standardization level and it is widely used in the context of the Link Open Data initiative. As tools evolved quite fast, the decision of which language to use may change at a later time.

6.2.2 Issues with Ontologies

In the discussion above we presented a definition of ontology, and then we presented three types of languages to describe ontologies, namely OWL and classification reasoning, SWRL and RIF for rule-based reasoning that goes beyond the classification, and finally SPARQL and SQWRL to query the Knowlege base and extract crucial information patterns. These three languages display the high potential for ontology reasoning, but ontology reasoning always come with two of important questions that need to be addressed, namely: the origin of the ontologies, or in other words who makes them? and how to deal with multiple ontologies that are potentially incompatible.

Origin of ontologies

In the original ideas that were behind the development of the semantic web, it was expected that complex ontologies would become available through major organizations, and especially standardization organizations, that would define the essential concepts in their domains. Some initial steps have been made in that direction with the standardization of SUMO[11] (Suggested Upper Merged Ontology) at IEEE. SUMO provides an "Upper Ontology" which provides all fundamental concepts that can be specialized for the different subdomains. Whereas there have been a number of specialized ontology developments, mostly related to the W3C[12] such as the *W3C Semantic Sensor Network* ontology, the uptake did not match the level initially hoped.

Nevertheless, in the last few years there have been a number of developments that show an uptake of Semantic Web technology and ontological reasoning. This development goes under the label of *Link Open Data*[13] (LOD) which produced billions of statements of semantically marked up data. The most recent map LOD is shown in Figure 6.1.

The size of LOD is quite impressive, consisting at this time of billions of statements, about virtually every possible topic. But despite its size, LOD displays two problem: one is that it is very heterogeneous and therefore difficult to use to address interoperability issues; the second one is that it is based on RDF(S), which is less expressive than OWL, and therefore less inferences can be made on it. Still, it provides an impressive amount of knowledge, and despite the fact that we did not exploit it yet, we are carefully monitoring its content and evolution.

Reference to different ontologies

The converse problem to the origin of ontologies is the availability of multiple alternative ontologies that are potentially inconsistent. The expectation that there will be one universal ontology that will provide all possible concepts in a coherent form is universally considered a dream. The experience with any attempt of standardization of data structures shows that the whole effort is very difficult even in very limited domains,

[11] http://www.ontologyportal.org/
[12] www.w3.org
[13] http://linkeddata.org/

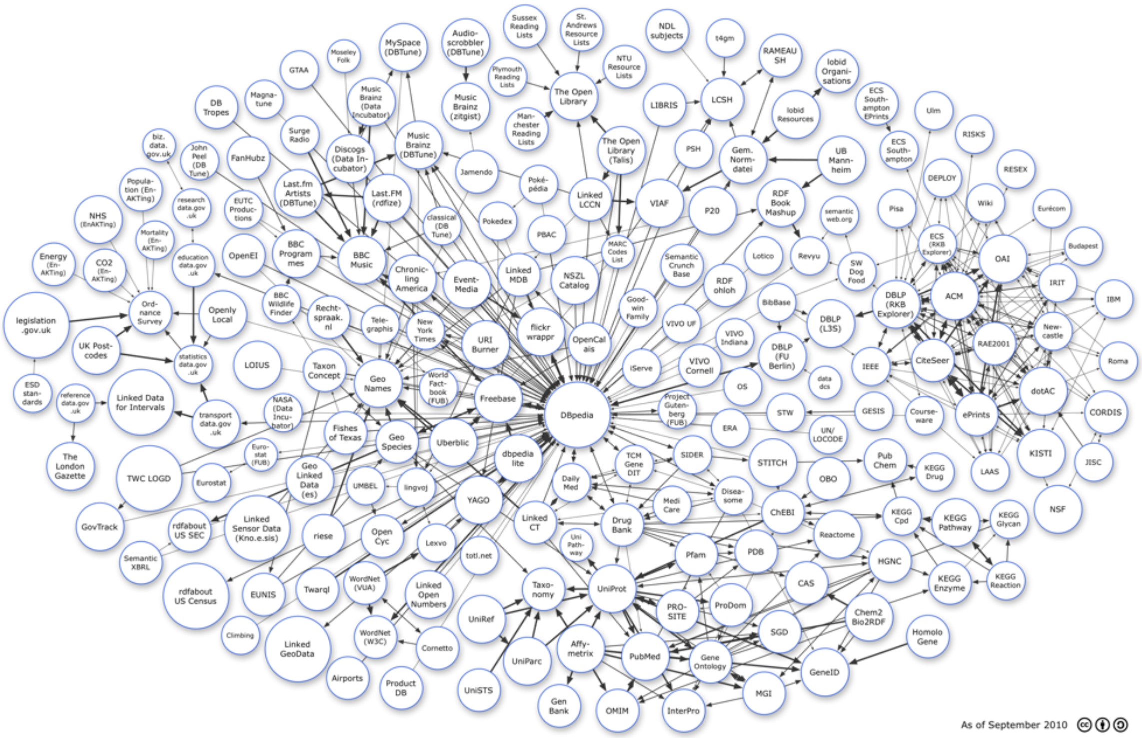


Figure 6.1: The Link Open Data cloud diagram. The bubbles correspond to ontologies, the links to the relation between ontologies

furthermore, the experience with distributed databases is that very different data structures emerge as the different database evolve. Therefore the likelihood that two services developed independently refer to the same ontology is quite limited.

The ideal way to address this problem is to construct an *alignment ontology*, such as SUMO above, which provides a way to relate concepts in the different ontologies. Essentially, the alignment ontology provides a mapping that translates one ontology into the other. Of course, the creation of an alignment ontology not only requires efforts, but more importantly, it requires a commitment so that the aligning ontology is consistent with both ontologies to be aligned.

When no alignment ontology exists, or when it is too expensive to build one, it would be ideal to build an alignment automatically. In the context of the semantic web there is a very active subfield that goes under the label of "Ontology Matching" [21, 59] which develops algorithms and heuristics to infer the relation between two concepts in two different ontologies. Essentially, the goal of the Ontology Matching effort is to construct the alignment ontology automatically. We already discussed the effort of Ontology Matching in the Deliverable D1.1 [6]. Ontology matchers use different resources, such as lexicographic indexes like WordNet to infer the relation between between two concept definitions in different ontologies. The results of the match is a relation and a level of confidence that that relation holds. Aside from the correctness of the results that can be derived, the major problem from our point of view is that the result of the match comes with a confidence value that specifies how related are the two concepts from the point of view of the ontology matcher.

The confidence value assigned by the ontology matcher is typically used to evaluate the quality of the results returned when the ontology matcher is used in the context of information retrieval. In this case the task is already essentially noisy, therefore there is a considerable likelihood that some of the information retrieved is irrelevant and conversely that some of the relevant information is missed. But in the context of constructing CONNECTors to resolve interoperability issues, the use of confidence immediately reflects on the confidence that we can assume on the CONNECTor. This means that we need to add a measure of utility of the match with respect to the task performed. For example, if the task is to relate two systems that provide weather information, the cost of an error for the use may not be that sizable; whereas if the task is to relate e-commerce systems where the cost of an error is that the user does not receive their goods or that some money is lost, then the cost of uncertainty is much greater.

In conclusion, the two concerns above, although mitigated, are still present in ontology engineering and its applications. Ultimately, ontologies come with a set of trade-offs that need to be explored more closely. In section 3.3.1 we already pointed out how ontologies can play a role in expressing affordances and therefore facilitating component discovery; furthermore, in Deliverable D3.2 [18] we showed how ontologies are employed to facilitate the synthesis of CONNECTors. In this chapter we will look more deeply at two other experiences that we made with ontologies, namely on the message mapping in the context of VANETs and on reasoning about system architectures in the context of the overall demonstrator discussed in Deliverable D6.2 [16].

6.3 Exploiting Ontologies to support Interoperability across VANETs

As a first experiment, we investigated the use of semantic technologies to tackle the problem of interoperability in a particular middleware domain, namely Vehicular Ad Hoc Networks (VANETs). In this domain, a numer of different protocols exist to route network messages between moving vehicles. These protocols are highly heterogeneous in terms of their routing strategies (broadcast-based, location-based, etc.,) and the packet formats employed, hence interoperability between protocols is a significant challenge. Here we demonstrate how ontologies can be applied to acheive better interoperability between VANET protocols; this highlights one of the key contributions of the CONNECT architecture, i.e., the use of ontologies at a deep level in the system to address the technical differences between low-level communication protocols. We argue that VANETS offer a good case-study for low-level interoperability in CONNECT because their overall goal is simple: to send a message from one node to another (and hence, we do not need to worry about other interoperability issues such as application data and behaviour differences). In the future, we will extend the approach to cover richer protocols such as RPC, service discovery and publish-subscribe.

At the heart of the approach is the construction of the domain's reference ontology, i.e. *the VANET ontology*, to build semantic understanding about the VANET protocols. This ontology acts as a repository, storing the descriptions of all of the different routing strategies available for this domain, and the different packet formats that result from them. We then investigated how the ontology could be utilised to perform two tasks important to achieving interoperablity:

- *Classification*. A reasoner, used within the ontology, enables the classification of observed packets under the appropriate routing category. In order to enable other packets to be routed, the new packet formats need to be compared against the existing packet formats in the ontology. For this to be possible, there needs to be a way to store the description of known packet formats and when new packets arrive, the system should be able to infer the description of the new packet formats and then allow the required comparison to take place. As we will show, to enable an adequate comparison, the ontology needs to be used in conjunction with SWRL rules and SQWRL queries. The SWRL rules enable a more expressive inference based on the information provided from the ontology itself. On the other hand, the LOD queries can be added in order to extract valuable information from this repository. For example, queries can be formulated to find fields that do not match between two different packet formats.

- *Mapping*. Classification can help build a semantic understanding of the packet content and the protocol behaviour, but it cannot determine how to map the data content from one message to another. We investigate here the use of SQWRL queries to underpin an approach to describe how two protocol messages can be mapped onto one another.

6.3.1 Vehicular Ad-hoc Networks

Vehicular Ad-Hoc Networks (VANETs) are an important type of ad-hoc network that are made up of vehicles interacting with each other. Their goal is to distribute information such as traffic data and road safety warnings in order to enhance the overall safety of the system. VANETs exhibit their own unique set of characteristics unlike other mobile ad-hoc networks, such as a high mobility feature and a very dynamic network. Figure 6.2 shows a vehicular network whereby the vehicles are communicating with each other in order to exchange messages.

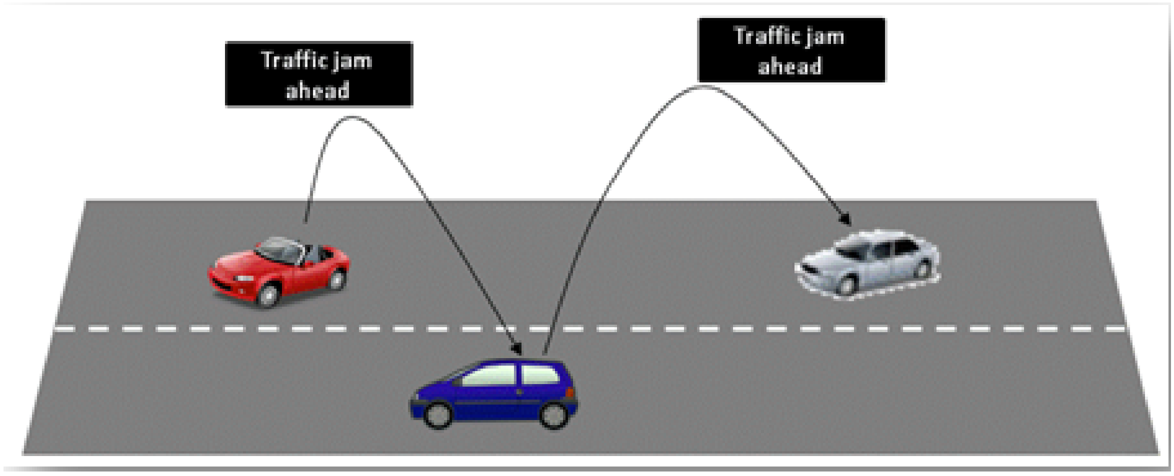


Figure 6.2: : Vanet Model

The success of VANET applications relies on the fact that messages should be easily distributed to different vehicles in the network and those messages should be easily interpreted. If all the messages follow a certain standard format, then there is not much problem in distributing the messages as all the vehicles will interpret them the same way. But this also means that the network is restricted and can only infer a certain type of message format. Hence, in order to enhance the scalability of the VANET, the network should be able to interoperate with other types of VANETs having different set of message formats. So far, different VANET systems are not able to interoperate with one another. To understand the concept of interoperability within VANETs, it is imperative to understand the different types of VANETs that exist. These different VANETs are formed based on the type of routing protocol they are employing in order to distribute messages.

Categories of Vehicular Ad-Hoc Network Protocols

There are different categories of routing protocols that are applicable in Vehicular Ad-hoc Networks (Vanet):

- **Broadcast-based**: The vehicles talk to each other in the network. When a message is broadcasted within the network, all the vehicles that are within the radio range receive this information.
- **Position-based forwarding** uses: node locations (longitude and latitude) and the greedy forwarding method in order to forward packets towards the node which is closest to the destination.
- **Trajectory-based forwarding** uses: the greedy forwarding method and an estimated trajectory outlined by the source node through GPS and a digital map in order to route the packets to the destination.
- **Restricted-directional flooding** uses: GPS or directional antenna in order to direct the message broadcast in the required direction. This method is useful in warning nodes behind the sender node e.g during occurrence of a road accident in front of the sender.
- **Content-based forwarding** uses: Distance, Speed, Traffic Density and availability of neighbours to route packets. The contents of packets are analyzed and based on the priority of the message, the packets are routed accordingly.
- **Cluster-based forwarding** uses: Clusters formed in the network where a head node is responsible to forward the messages to the other nodes in the cluster. A border node is assigned to route the packets outside of the cluster.

- **Opportunistic forwarding** uses: the Carry and Forward notion in order to route the messages in the network.

Example of routing protocol messages

This section demonstrates a few packet formats that are used in the context of VANETs; thses are: BBR (for Broadcast-based VANETS), Broadcomm (for Trajectory-based VANETS) and Lora_cbf (for Location-based VANETS). The idea is to show how VANETs emitting different types of packets can interoperate with one another. In Figure 6.3 we see example of the packet formats for each of these three protocols.

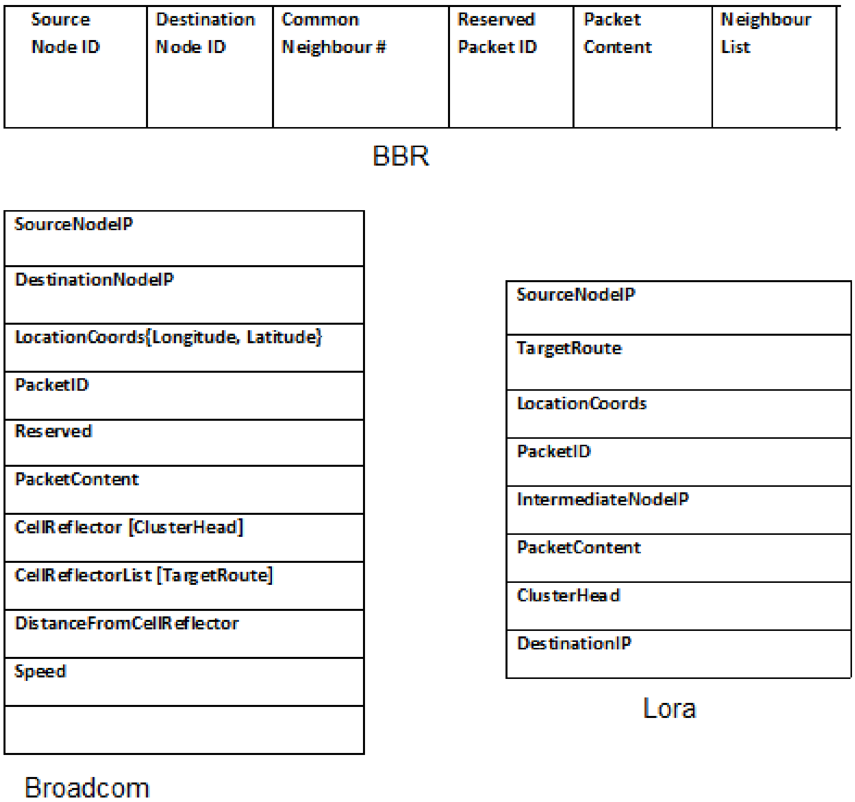


Figure 6.3: Packet formats of three VANET Routing Protocols

Figure 6.3 shows that a direct mapping of packets from one type of VANET to another type of VANET is not possible. In order to achieve interoperability within different VANET protocols, we need to learn the behaviour of each protocol. Then, based on this information, we need to try to find some degree of similarity (matching), if possible, between these two different protocols before any procedure of mapping is applied.

As mentioned previously, the ontology can be used to relate information between similar sources and help achieve interoperability among these systems. If we take, as an example, a Broadcast-based packet (e.g. BBR) and a Location-Based packet (e.g. Lora_cbf), can an ontology really help in enabling an exchange of information between these two different packets? Figure 6.4 highlights these two different packet formats. The Broadcast-based VANET requires only the DestinationIP to route the packet whereas the Location-based VANET requires both the DestinationIP as well as the GPS Coordinates. Hence, the location based packet can be used by the broadcast protocol, but the location-based protocol requires extra information, i.e. GPS co-ordinates, to be added to the broadcast packet in order for it to use it.

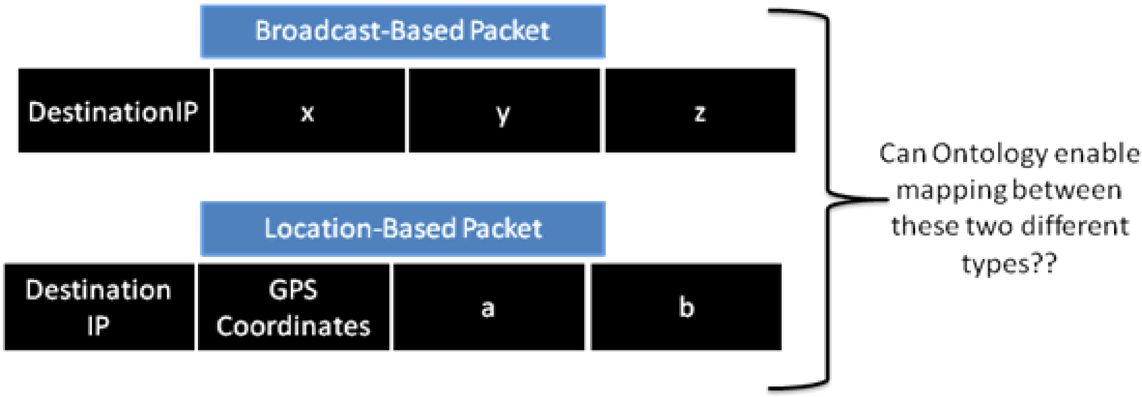


Figure 6.4: Comparing VANET Packet Formats

6.3.2 Application of ontologies to the VANET domain

As an exercise and also in order to answer the questions from the previous section, an ontology has been created in order to represent the domain of the VANET. The purpose of the ontology is to represent all the concepts that form the VANET, in particular the different types of packet formats that are possible for the different types of routing strategies applicable to this type of network. Figure 6.5 shows a snapshot of the VANET ontology; this shows the different concepts represented as classes. For instance the class Packets contains subclasses among which we have **ClusterBasedPacket**, which represents packets performing **Cluster-based routing**. The given classes also have fields such as a Cluster Head, a Target Route and some Location Coordinates.

Before the reasoner can be applied to achieve *classification*, we need to create a set of classes called *Restrictions* within the ontology. The purpose of these restrictions is to define new concepts which will be used to categorise the existing concepts. For example, we have created the definitions of packets like **BBR**, **Broadcomm** and **Lora_cbf** within the ontology. We require the reasoner to classify these packets under different categories of VANETs, for example, BBR should be classified as a Broadcast-based packet. Broadcomm should be classified as Trajectory-based packet but can also be classified as a Partial-LocationBased packet since it also contains part of the set of required fields to operate as a Location-Based packet. All these different categories are defined as restrictions within the ontology and the purpose of the reasoner is to classify the different packets under these different restrictions, based on the definitions provided by the packets.

When the reasoner is applied, the inferences generated are displayed within the ontology itself as demonstrated in Figure 6.6. This reveals the inferred concepts from the VANET ontology. For instance, the Broadcomm packet has been classified under the class **PartialPositionBased** packet. This class describes all classes of packets which contain part of the set of required fields to operate as a Position-based packet (these fields are location coordinates and a trajectory). Since Broadcomm packet contains either of these required fields, it is therefore classified as a PartialPositionBased packet.

Weakness of Ontology. So far, the ontology is able to classify the packets as per the description provided by the VANET domain itself. However, even if Broadcomm, for example, is found to be a partial Position-based packet, the ontology cannot tell what fields are lacking from the Broadcomm packet for the latter to function as a fully Position-based packet such as Lora_cbf. This is where the ontology lacks expressivity. In order to fulfill this mapping need, we need an extra mechanism that can increase the expressivity of the OWL language, which are *SWRL rules*.

Using OWL it is possible to to classify the packets as per the description provided by the VANET domain itself. However, even if we can tell that Broadcomm, for example, is found to be a partial Position-based packet, the ontology cannot tell what fields are lacking from the Broadcomm packet for the latter to function as a fully Position-based packet such as Lora_cbf. This is where OWL lacks expressivity. In order to fulfill this need, we need to increase the expressivity of the OWL language with the SWRL rules.

Figure 6.5: VANET Ontology

6.3.3 Using SWRL

In Broadcast-based VANETs, packets are broadcasted a certain number of times, depending upon the density of the network in order to reach a maximum number of vehicles. Hence, if we want to input into the ontology that a certain packet has a high broadcast rate (the class representing this is termed as hasHighBroadcastMeter), we can set up a SWRL rule as follows:

```
BroadcastMeter(?b) ^ BBRPacket(?p) ^ hasIntegerType(?b, ?s) ^
            swrlb:greaterThan(?s, 10) -> hasHighBroadcastMeter(?p, ?b)
```

This rule states that if a packet p is a BBR packet, and has a broadcast rate b (BroadcastMeter) and this broadcast rate b has value s, and if the value s is greater than 10 (a threshold value), then it implies that the BBR packet b has a high broadcast meter.

So far, the ontology and the reasoner can classify a packet under a certain category. For example, BBR packets are classified under Broadcast-based Vanets and Lora_cbf packets are classified under Position-based Vanets. In addition, the rules help to increase the expressivity of the ontology as shown above. However, if we need to compare these two types of packets and determine which fields are different

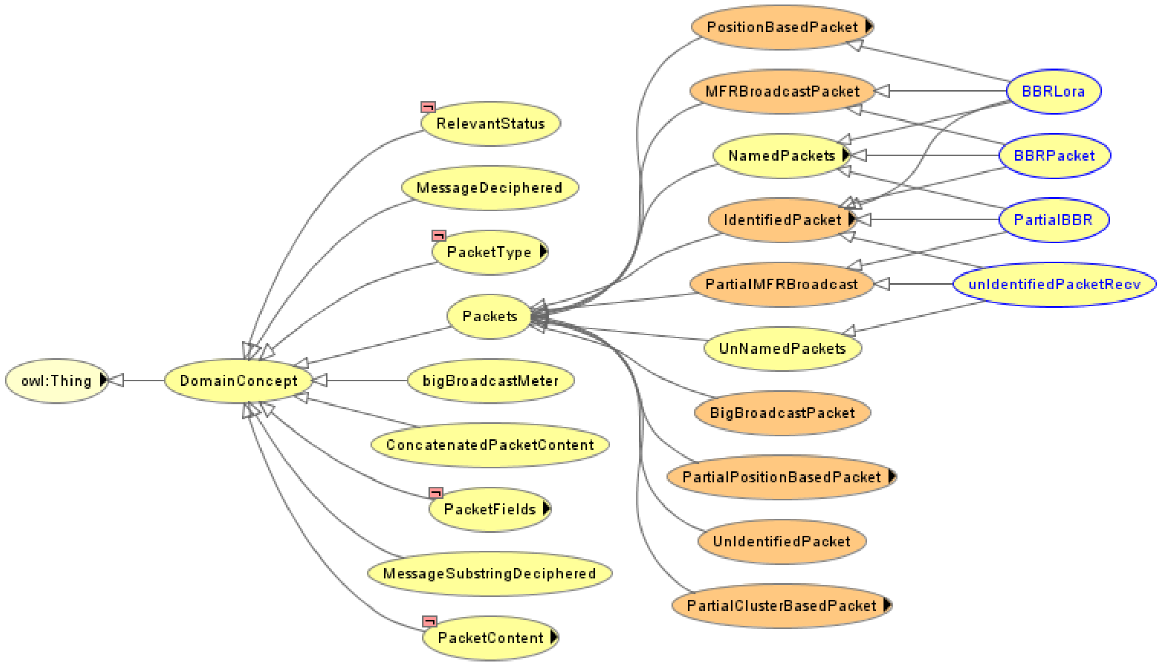


Figure 6.6: Inference in VANET Ontology

between them, we cannot do so with the ontology and SWRL rules only. Henceforth, we need to make use of *SQWRL*.

6.3.4 Using SQWRL

An an example, suppose we need to compare the packets BBR and Broadcomm and find out the fields which are different between them. The SQWRL query will look as follows:

```
BBRPacket(?b) ˆ hasFields(?b, ?f) ˆ Broadcomm(?p) ˆ hasFields(?p, ?pf) ?
        sqwrl:makeBag(?bag, ?f) * sqwrl:makeBag(?bagt, ?pf)
       * sqwrl:difference(?diff, ?bagt, ?bag) ˆ sqwrl:element(?e, ?diff)
        -> sqwrl:selectDistinct(?p, ?e)
```

For instance, the SQWRL query above states that if b is a BBR packet and has fields f, create a bag or set of all these fields called bag. On the other hand, if p is a Broadcomm packet and has fields pf, create a set or bag of such fields called bag t. Then find the difference between these two bags and if there is any difference, then select those fields which are found to be in Broadcomm packet p but not in BBR packet b. The result of this query is the set of fields found in Broadcomm and not in BBR.

The OWL language enhanced with the use of SWRL and SQWRL results in creating a very expressive VANET ontology which can determine the nature of a packet given the field descriptions, and also enable the comparison of any two particular packets and to finally produce the fields that are different between them. Once the matching of the packets has been achieved, this leads to the next step in this interoperability approach which is to perform the mapping between these two packets.

6.3.5 Enabling mapping in VANET

Referring to the above scenario, if a Location-based VANET receives a BBR packet, the fields that are missing from the latter are: `Location Coordinates`, `Target Route`, `Cluster Head`, `Distance From Cell Reflector`, and `Speed`. In this case, the mapping algorithm should map the list of Neighbours received from the BBR packet onto the Target Route (which is a list of possible destinations for routing the packet). It should further decide a cluster head for this particular case. Moreover, if the hello beacons coming from the source of the BBR packet contain Location Coordinates, they can be used to fill in the gap. But if the Location coordinates are missing, then the routing algorithm should compute a best possible range of Location Coordinates for this particular vehicle, based on the location coordinates of the neighbours and still route the packet using a range of Location Coordinates instead of an exact pair of Location Coordinates.

On the other hand, when a Broadcast-based VANET (using BBR packets) receives a Broadcomm packet (a Trajectory-based packet), the fields that are necessary for the Broadcast-based VANET to function are: `Source Node ID`, `Destination Node ID`, `Common Neighbour #`, and `Neighbour List`. Among these fields, the Broadcomm packet can only provide Source NodeID and Destination NodeID. In order for the Broadcast-based VANET to accommodate the Broadcomm packet, it would need the lacking two fields which are Common Neighbour # and NeighbourList. One possible solution to this problem is to create an adaptive routing strategy so that the routing can still be performed even if few fields are found to be missing. In a VANET, vehicles discover their neighbourhood by sending and receiving hello beacon messages. Therefore, the Broadcast-based VANET can infer the list of neighbours for the Broadcomm packet by looking at the possible beacons received within the same timeframe as the latter, and hence also approximate a possible common neighbor number.

6.3.6 Analysis

We have presented one example of a mapping procedure being applied in the case of VANETs. Interoperability between protocols that employ different routing algorithms is underpinned by the need to identify the fields which are crucial for performing the routing of a particular packet format. Therefore, there is a need to cater for situations where those fields can be missing and provide an adaptive mechanism to counter this problem. We have shown the important role ontologies can play in moving The role of the ontology coupled with SWRL and SQWRL rules is to produce the set of fields that are found to be missing from a packet which has been received by a VANET using a particular packet format (we have shown here that this can be achieved). The next step is to enable an adaptive mapping mechanism in order to enable interoperability; work in the third year of the project will focus on this particular issue.

6.4 Using Ontologies to model System Architectures

In the previous section we used ontologies to reconcile low level details in the message structure of different protocols. In this section, we use ontologies to analyze systems at the architectural level. Specifically, we look at the one of the demonstrators implemented in the CONNECT project, where we are working towards the connection of two very different systems: the RCS MMIM: a multi-media instant messaging system, that is a system for multi-media distribution developed within the mobile network [30], and a video surveillance system, with the objective to improve video surveillance by including videos coming from different users.

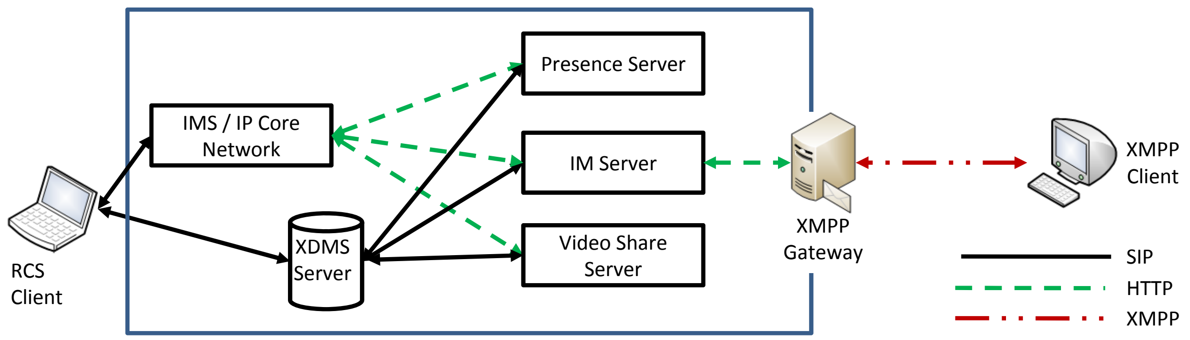


Figure 6.7: The two alternative ways to address the integration of the RCS MMIM and the Video-Surveillance system

As shown in Figure 6.7, there are two ways to achieve this result:

1. The mechanism for video-surveillance could open a connection directly as an RCS client, as the client shown on the left of Figure 6.7; but there is a complication that the RCS is restricted to mobile users, therefore the video-surveillance cameras should also register as users with the core of the mobile network and carry a SIM card credentials.

2. Exploit the non-standardized XMPP port, shown on the right of Figure 6.7, offered by the specific RCS system to which we have access. In this case though, any client to the XMPP port should be inside the domain in which the XMPP port is deployed.

In either case, the problem of connecting the two systems does not depend only on the protocol of the two connecting systems, but it also depends on additional conditions that need to be satisfied in the overall system. We need therefore a way to express: (1) the requirements of the different ports, for example, the need for the credentials on one side and the domain requirements on the other side; and (2) the capabilities of the clients, whether they have the credentials, and whether they can accommodate the domain requirements.

Ontologies afford us a way to address these problems. Indeed, we can specify ontologies that describe the different components, their relations, and their requirements, as well as the capabilities of the clients. We can then use the inferencing to derive whether the particular client fits the requirements of the different ports. Below we provide a brief description of how ontologies have been used to address the domain problem, and in Deliverable D6.2 [16] there is a more extensive description of the process.

A fragment of the relevant ontology is given in figure 6.8. We defined two main concepts: *Pattern* and *Component*. Components specify the type of components in the system, while Pattern describes their relation in the system. Here a system component is defined as having a *Domain* and a *Protocol*. Furthermore, two different types of components are identified: *Client* and *Server*, and components may be in relation among through the relations *hasClient* and *hasServer*.

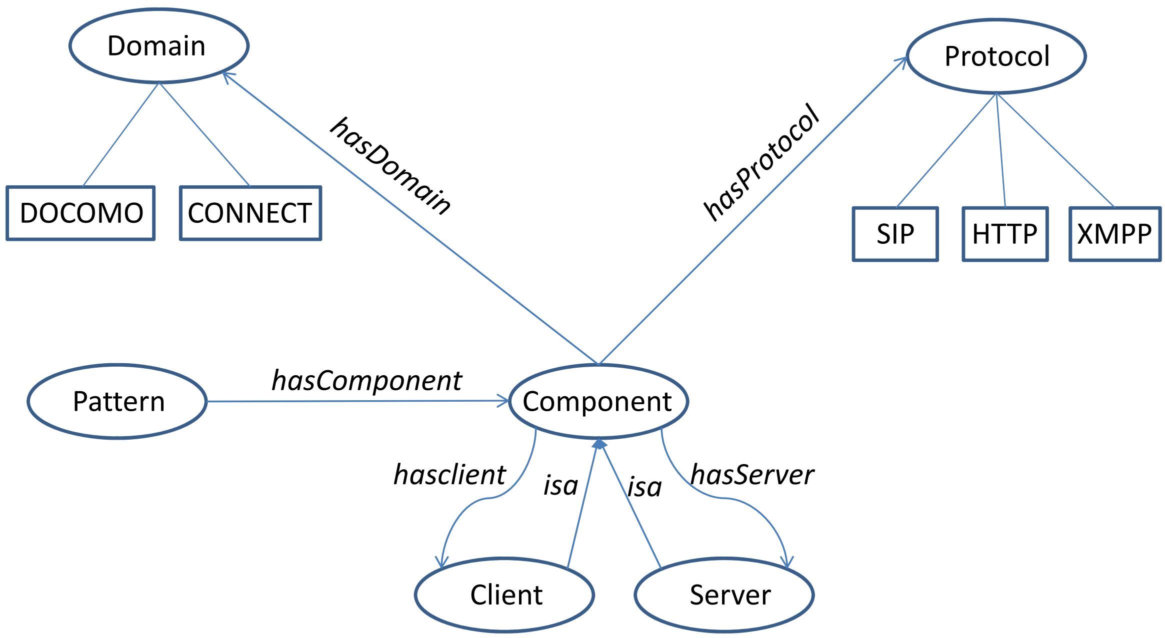


Figure 6.8: The different types of components in the ontology and their relation

Given the definition of Component, we can define patterns of components. In the context of the definition of the ontology that we are specifying we can take components and check in what type of pattern do they fit in. A fragment of the specification of patterns is shown in Figure 6.9. Patterns relate only to components, where each Pattern may relate to multiple components. It is therefore easy to define *Client/Server* patterns, as well as *ServerServer* patterns by restricting on the number of components that can participate in a Pattern and on the type of components that can be in a pattern. Furthermore, it is possible to identify different patterns that are problematic, as for example a pattern with clients only. Finally, it is possible to define a set of remedial actions, which then can be taken either automatically by the code or by some other operator to fix the problem.

The need for verifying when the system that is created by the connector is sound or not was required to check whether there are unexpected domain boundaries that should not be violated. As shown in Figure 6.7, both the client of the RCS XMPP port should be within the DOCOMO XMPP domain, and any client outside such domain would fail to communicate. The natural way to model such a constraint would be to define the patterns of components that are in the same domain, identified in Figure 6.9 as *SameDomain* and the ones that cross domains, identified as *DifferentDomain*. We could then define the class of patterns that are problematic and derive an appropriate remedial action, such as creating a connection through an intermediate XMPP server recognized by the RCS server.

As in the case of VANETs above, OWL is not enough to express all the constraints that are required by this case. For example, the definition of *SameDomain* and *DifferentDomain* cannot be expressed within the OWL language. But OWL in combination with SWRL and a query language provides us with the required expressivity.

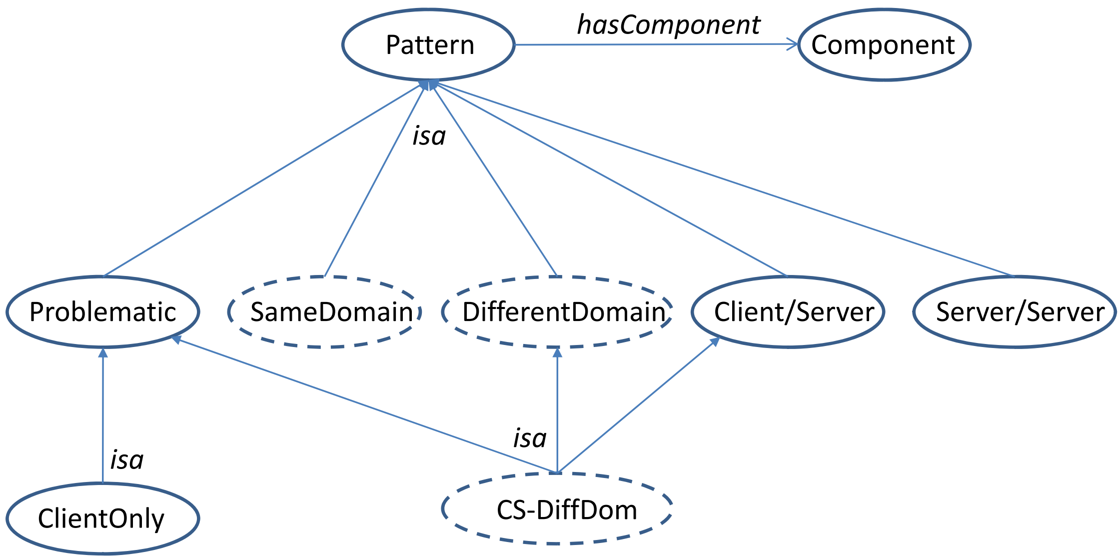


Figure 6.9: A snitppet of the Pattern Ontology.

6.5 Analysis: Towards an ontology of Middleware

In the introduction of this section about ontologies, we argued that the use of ontologies is very appealing because of their capability to perform inferences over system models. Ultimately, the broad use of ontologies in CONNECT proves that to be the case. Within the CONNECT project we have used ontologies to model five different aspects of systems of systems: (1) to analyze the message structure as described above in the context of VANETS in section 6.3; (2) to perform Input/Output Matching and (3) to match systems affordances during discovery; (4) to model middleware to match different protocols; and (5) to analyze the structure of systems architectures. These experiences provide us with a broad range of uses of ontologies at different levels of abstraction in the systems analyzed.

When reviewing these experiences it is quite clear that ontologies essentially provide us two functions that would be difficult to build otherwise.

1. *Abstraction over syntactic differences of systems descriptions.* This is evident with respect to affordance matching. There, two systems may use different labels to describe their own affordance, but when the labels are substituted with concepts defined with respect to an ontology, then the proof theory can be exploited to verify the relation between these concepts to find the degree of matching.

A similar process was exploited for the Input/Output Matching during Ontology-based Model Checking in section 3.3.2. In this case the reasoning hinged on proving whether two syntactically different traces can, or cannot be considered equivalent through an analysis of the relation between inputs and outputs. In a sense, this reasoning provides the first step beyond the reasoning about message structure to look also at the payload of the messages carried.

2. *Reason about the structure of objects that need to be manipulated, their features and their missing features.* This is particularly clear in the analysis of message structures, which was analyzed in the context of VANETS in section 6.3. There ontologies proved useful to identify the types of messages and the additional requirements that need to be satisfied to transform across different types of messages, and what is missing in a message to transform it to the other type of message. A similar resoning problem was presented by the verification of architectures, as discussed in section 6.4. There we showed that, even when all the pieces are in place, additional problems may emerge from the point of view of architectural requirements that are not satisfied. The verification of these architectural requirements requires reasoning about the relations between the different components and whether the conditions for these components to work together are satisfied. As in the previous case, it is not enough to verify whether the CONNECTed system architecture is valid, but also how to fix it, and in turn what is missing to transform it to a valid architecture.

Ultimately, the abstraction over syntactic differences of systems descriptions provides us a semantic abstraction with respect to an ontology. As a result, the ontology allows us to recognize when two structures that look superficially different from the viewpoint of their syntactic structure carry essentially the same type of information, and it is therefore possible to compare and analyze them in a coherent way. Furthermore, the ability to reason about the structure of entities that need to be manipulated, their features and their missing features, allows us to express in a declarative way the entities that are involved in the systems that need to be connected, and reason about the connected system and very different levels of abstraction, from the message structure to the hight level architecture.

6.5.1 Modeling Systems with Ontologies

Ontologies have been used quite extensively to represent aspects of systems, but as far as we could see, there has never been an ontology quite as broad and comprehensive as the one emerging in CONNECT. From the point of view of CONNECT, we are interested in work that addresses the two problems highligted above: namely the abstraction from the syntactic aspects of the system description; and, second, the reasoning about systems.

As for the first problem, the most prominent work in literature have been developed around semantic web services [42, 22, 65], which were reviewed extensively in Deliverable D1.1 [6]. This work has been the basis of the representation of affordances of the representation of traces. Though, most of these works concentrate on a description of the affordance as a function mapping inputs into outputs, while in CONNECT we have a broader approach in which affordances are described to reflect the function performed by the system independently of the input/output relation that they generate. The second aspect is that the discovery extends to the mapping of traces of protocols which are typically not considered in semantic web services.

On the second problem, namely reasoning about systems, there have been two main streams of work. On one side, Description Logics has been used to provide a formal semantics to UML [5, 60, 36] with the objective to verify their consistency and possibly debug them when they are invalid. The second stream of work has been toward the analysis of the configuration of systems. The problem here is that system configuration tends to become a combinatorial problem and a logic model can help the developer to avoid any misconfiguration. The first attempt in this direction has been due to [4] whose system was successfully applied to the configuration of very complex telecommunication systems at AT&T and Lucent. The configuration problem has been considered in other systems which applied both to computer systems as well as other systems such as installation of home electronics [45].

Our work closely relates to these works, and indeed, it may be beneficial to revise our ontology to reflect UML more closely. On the other hand, our problem is somewhat different since we do not aim at an operator that has to verify the settings of an apparatus, or the validity of UML diagrams, rather we aim at developing a system that constructs correctly a CONNECTor between two other systems. Therefore, it is not enough to detect the failure, we need also to identify it and react to it. Indeed, in our ontology, shown in Figure 6.9, we have the concept “Problematic” that is the root node of the possible faults that we can have in the system and we can react to.

In perspective, ontologies may afford another level of abstraction in building valid CONNECTors. Rather than specify all possible faults and keep a fixed relation between the faults and the possible fixes, we could use the inference engine to prove whether the overall system is valid. If the proof fails, we could analyze the proof to derive the fault, and potentially create a fix for it on the fly. Such a fix would describe how to fix the system to generate a correct CONNECTor. Some technologies to support this process are already available: work on proof explanation [19] could provide a way to derive the reason of a proof from which to derive the system fault, and we could exploit AI planning technology [27, 51, 61] as a mechanism of automatic programming to generate a system fix for us. But at this point the generation of explanations is still very difficult and possible only for very weak logics; as for AI planning, it is not clear how to do planning with actions at different levels of abstraction. In the cases analyzed by CONNECT so far, there is no need for such complexity. Although this approach sounds too visionary even by CONNECT standards, we may investigate this opportunity in the progress of the project work.

6.5.2 Issues with Modeling

In the presentation so far we have looked at ontologies in an abstract form, but it is important also to look at the practical consequences of using ontologies for system modeling. Specifically, the use of OWL as a modeling language comes with two types of problems. On one side, since the standards are still evolving, the tools are not quite at the same level as the standards; on the other side, the use of logics requires a mind shift with respect to traditional programming.

With respect to the standards and the tools, we noticed above as SWRL is being replaced with RIF, and ad hoc query languages such as SQWRL are replaced with SPARQL, with the problem that tools have to be modified to adapt to the standards shifts. This is evident in *Protege*, which is now the leading open source software to create ontologies. Recently Protege did undergo an extensive rewriting moving from version 3, which supports OWL1.1 SWRL , to version 4, which supports OWL 2.0 and is tailored to handle large and complex OWL ontologies, but has no support for SWRL and SQWRL.

The second problem is that OWL and DL in general require a change of thinking that is sometimes inconsistent with more traditional programming languages. The main problem is that it makes the *open world assumption* which means that if something is not told it cannot be assumed to be false. The use of the open world assumption allows the use of OWL with incomplete and evolving information, but it is also contrary to traditional programming languages that typically make a closed world assumption. In practice the open world assumption forces the ontology developer to define a number of negative statements that are quite unintuitive to make. The second problem is that it does not make the *single name assumption* therefore it is often difficult to assume that two things that appear to be the same are actually the same.

6.6 Conclusion

The result of the work performed during the year is that we have developed ontologies that describe different aspects of systems: from the low level details of their messages, to the payload of those messages, to their affordances, to the abstract architecture. As a result, we are progressing toward the objective to build a CONNECT ontology of middlewares which is broad enough to express very different middleware structures as well as very different aspects of computational systems.

The ontology that we have now is quite broad, but also fragmented in the sense that it evolved in an ad-hoc way while the project members were addressing different problems. This result has been the consequence of a decision to develop ontologies “bottom up” in which ontologies are first used to solve problems, rather than following a ”top-down” approach where ontologies are created in an abstract way and then used to impose a structure on the problem solving, with the risk of missing the crucial concepts that we need and of creating artificial problems. The work to be performed next year will be to synthesize those ontologies in a coherent and, as much as possible, complete language to describe systems. Furthermore, we will be able to extend our ontologies to represent other aspects of systems such as quality of service, security policies, and reliability constraints.

7 Conclusions

7.1 Concluding Remarks

The overall aim of the CONNECT project is to bridge the interoperability gap that results from the use of different data and protocols by the different entities involved in the software stack such as applications, middleware, platforms, etc. This aim is particularly targeted at heterogeneous, dynamic environments where systems must interact spontaneously i.e. they only discover each other at runtime. In this document, we have presented a refined version of the CONNECT architecture that will meet this particular aim; this focuses on producing and integrating concrete technologies in order to maker the vision a reality.

In this report we first presented a refined version of the CONNECT architecture. This highlighted the concrete vision of the CONNECTor architecture and the Networked System Model; both of which play a central role in integrating the work of the separate work packages. Also, the enabler architecture and the associated communication exchange between enablers was unified within the CONNECT architecture.

A number of prototype software solutions have been created on the path towards the full implementations of the CONNECT architecture. Those highlighted in this deliverable and described in the accompanying appendix (Deliverable D1.2 Appendix - Prototype) are:

- *The Discovery Enabler* protoype can discover and generate the networked systems deployed in a network environment, for systems that have been advertised by UPnP or the CONNECT Discovery Protocol. Further, the software matching framework can identify networked systems that match and are suitable to interoperate with one another. Further information about this prototype is found in Section 2 of the D1.2 Appendix - Prototype.

- *The deployment enabler* can deploy CONNECTors in the network environment to successfully interoperate with the legacy networked systems that are also deployed. Further information about this prototype is found in Section 3 of the D1.2 Appendix - Prototype.

- *The* CONNECT*or generator tools* allow software CONNECTors to be realised from the high-level models produced (or defined) by other elements of the CONNECT architecture. The prototype software corresponding to these tools is collectively termed the Starlink framework and further description including the location is given in Section 4 of the D1.2 Appendix - Prototype.

These prototypes have been or are being evaluated using small application or middleware case-studies; these show that the prototypes succesfully achieve the objectives of the CONNECT project (namely long-lived interoperability) and conform to the specification of the CONNECT architecure. The prototypes also form the foundation of the implementation and evaluation of the case-study as described in Deliverable D6.2 [16].

The other important contribution of this report is to highlight the important role of ontologies in the CONNECT architecture. Ontologies have been successfully employed within Web 2.0 applications, however these have only really considered the top level concerns such as discovering semantically similar systems. CONNECT is pushing the role of ontologies further, we are going deep with our use of ontologies, using them across system software as well as at the application level. Our ontologies cross-cut all of the CONNECT functions and enablers. We have shown the role of ontologies in the discovery, matching, and synthesis of CONNECTors; here ontologies feature in the networked model and are employed in discovery and matching of affordances and descriptions, further, matching of systems (including alignment based upon ontologies) leads to the synthesis of CONNECTors. At the lowest level of the CONNECT architecture, i.e. the interoperation with middleware protocols, ontologies are applied to classify (discover the behaviour of) new network protocols and are used to determine the low-level interoperability bridges (i.e. matching and mapping of data field content from between messages).

Overall, this intermediary step has proven that the advanced solutions proposed by CONNECT can be integrated and produce appropriate interoperability solutions that ensure heterogeneous networked systems can communicate. The nature of this process means that there remain further open questions and challenges as we seek to create eternal and long-lived interoperability software.

7.2 Future Activities for WP1

There are two important areas of future work: i) the addition of advanced learning and synthesis approaches into the CONNECT architecture; ii) addressing non-functional properties through the integration of the dependability enablers (as provided by WP5). These will form the body of work for WP1 in the third year of the project, with the key objective of achieving fully automated, future-proof and dependable interoperability solutions, and hence achieve the overall goal of the CONNECT project.

In terms of *advanced learning*, we envisage further investigation of the role learning occupies within the architecture. At present, learning is focused solely on the behaviour model from the networked system model; that is, it aims to identify the application behaviour of a system. While this is important to the automation of CONNECTors, it only focuses on part of the behaviour. At present, the middleware protocol behaviour and their corresponding message formats must be defined (and be known by CONNECT) in advance. If a new system employs a novel protocol then CONNECT is unable to resolve the interoperability, hence the approach is not future proof. Rather it is required that we equally apply learning approaches at the middleware level; this would not be executed as frequently (e.g. within the flow of the CONNECT process) because a new protocol need only be learned once. At present synthesis is disjoint, we can match and synthesize solutions at the separate levels of application and middleware. We will integrate the work about *advanced synthesis* techniques from WP3 that synthesize complete CONNECTor solutions into the CONNECT architecture.

For the *non-functional properties* and *dependability assurance*, we plan to integrate the work of WP5 (as discussed in D5.2 [15]) into the prototype software of the CONNECT architecture. In particular, this will first involve eliciting the non-functional requirements from networked systems and specifying them using the property meta-model within the Networked System Model. This will involve extending the discovery enabler to discover information about the systems and the environment in order to describe these non-functional properties. The further steps will involve integrating the dependability and monitor enablers to ensure that runtime dependability analysis is achieved for the deployed CONNECTors. Finally, we will investigate concrete solutions to add the body of work about trust and security into the architecture as a whole.

8 D1.2 Appendix

8.1 xDL definition of the Photo Sharing Networked Systems

```
<?xml version="1.0"?>
<xDL name="LimePhotosharing"
        targetNamespace="http://example.com/LimePhotosharing.xdl"
        xmlns:tns="http://example.com/LimePhotosharing.xdl"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns="http://www.connect-forever.eu/xDL>
  <types>
    <type name="PhotoMetadata"
          modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#
              PhotoMetadata"
        liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
            photoMetadata2Ont.xslt"
        loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
            ont2photoMetadata.xslt">
      <xsd:complexType>
        <sequence>
          <element name="photoID" type="string"></element>
          <element name="location" minOccurs="1" maxOccurs="1">
            <complexType>
              <attribute name="latitude" type="double"></attribute>
              <attribute name="longitude" type="double"></attribute>
            </complexType>
          </element>
          <element name="details" type="string"></element>
        </sequence>
      </xsd:complexType>
    </type>
    <type name="PhotoMetadataList"
       modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#
           PhotoMetadata"
       liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
           photoMetadata2Ont.xslt"
              loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/
                  mapping/ont2photoMetadata.xslt">
      <element name="metadataElt" type="tns:PhotoMetadata" > </element>
    </type>
    <type name="PhotoFile"
          modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#
              PhotoMetadata"
          liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
              photoMetadata2Ont.xslt"
        loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
            ont2photoMetadata.xslt">
      <xsd:complexType>
        <sequence>
          <element name="photoID" type="string"></element>
          <element name="File" type="hexBinary"></element>
        </sequence>
      </xsd:complexType>
    </type>
    <type name="PhotoComment"
          modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#
              PhotoComment"
          liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
              photoMetadata2Ont.xslt"
          loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/
              ont2photoMetadata.xslt">
      <xsd:complexType>
        <sequence>
          <element name="photoID" type="string"></element>
          <element name="comment" type="string"></element>
        </sequence>
      </xsd:complexType>
```

```
    </type>
    <type name="PhotoIDTemp"
          modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#PhotoMetadata"
          liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/photoIDTemplate2Ont.xslt"
        loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/ont2photoIDTemplate.xslt">
    <xsd:element name="photoID" type="string"></element>
    </type>
    <type>
      <template name="PhotoDetailsTemp"
         modelReference="http://www.connect-forever.eu/ontologies/applications/photoSharing.owl#PhotoMetadata"
         liftingSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/PhotoDetailsTemplate2Ont.xslt"
         loweringSchemaMapping="http://www.connect-forever.eu/ontologies/applications/mapping/ont2PhotoDetailsTemplate.xslt">
      <element name="details" type="string"></element>
    </type>
  </types>

  <primitives>
    <SMPrimitives>
      <write name="writePhotoMetadata" binding="tns:LimeOutPhotoMetadata">
        <sdata name="SPhotoMetadata" type="tns:PhotoMetadata"/>
      </write>
      <write name="writePhotoFile" binding="tns:LimeOutPhotoFile">
        <sdata name="SPhotoFile" type="tns:PhotoFile"/>
      </write>
      <read name="readPhotoComment1" binding="tns:LimeRdPhotoComment1">
        <template name="PhotoIDTemplate" type="tns:PhotoIDTemp"/>
        <sdata name="PhotoFile"/>
      </read>
      <read name="readPhotoComment2" binding="tns:LimeInPhotoComment2">
        <template name="PhotoIDTemplate" type="tns: PhotoIDTemp"/>
        <sdata name="PhotoFile"/>
      </read>
      <write name="writePhotoComment" binding="tns:LimeOutPhotoComment">
        <sdata name="SPhotoComment" type="tns: PhotoComment"/>
      </write>
      <read name="readPhotoMetadata" binding="tns:LimeRdgPhotoMetadata">
        <template name="PhotoDetailsTemplate" type="tns: PhotoDetailsTemp"/>
        <sdata name="SPhotoMetadataList" type="tns: PhotoMetadataList"/>
      </read>
    </SMPrimitives>
  </primitives>
  <bindings>
    <binding name="LimeOutPhotoMetadata">
      <data> <lime:data use="encoded"/> </data>
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl#out"/>
      </primitive>
    </binding>
    <binding name="LimeOutPhotoFile">
      <data> <lime:data use="encoded"/> </data>
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl#out"/>
      </primitive>
    </binding>
    <binding name="LimeRdPhotoComment1">
      <data> <lime:data use="encoded"/> </data>
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl#rd"/>
      </primitive>
    </binding>
    <binding name="LimeInPhotoComment2">
      <data> <lime:data use="encoded"/> </data>
```

```
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl
          #in"/>
      </primitive>
    </binding>
    <binding name="LimeOutPhotoComment">
      <data> <lime:tuple use="encoded"/> </data>
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl
          #out"/>
      </primitive>
    </binding>
    <binding name="LimeRdgPhotoMetadata">
      <data> <lime:data use="encoded"/> </data>
      <primitive> <lime:action use="http://www.connect-forever.eu/ontologies/middleware/lime.owl
          #rdg"/>
      </primitive>
    </binding>
  </rbindings>
</xDL>
```

8.2 BPEL Behavior of the Photo Sharing Affordances

```
<bpel:if name="If">
  <bpel:flow name="Flow">
    <bpel:sequence name="Sequence">
      <extensionActivity>
        <xdl:write name="writePhotoMetadata">
          <data name="PhotoMetadata"> photoMetadata </data>
        </xdl:write>
      </extensionActivity>
      <extensionActivity>
        <xdl:write name="writePhotoFile">
          <data name="PhotoFile"> photoFile </data>
        </xdl:write>
      </extensionActivity>
      <bpel:while name="While1">
        <bpel:flow name="Flow1">
          <extensionActivity>
            <xdl:read name="readPhotoComment1">
              <template name="PhotoIDTemp"> photoID </template>
              <data name="PhotoFile"> photoComment </data>
            </xdl:read>
          </extensionActivity>
          <bpel:sequence name="Sequence">
            <extensionActivity>
              <xdl:read name="readPhotoComment2">
                <template name="PhotoIDTemp"> photoID </template>
                <data name="PhotoFile"> photoComment </data>
              </xdl:read>
            </extensionActivity>
            <extensionActivity>
              <xdl:write name="writePhotoComment">
                <data name="PhotoComment"> comment </data>
              </xdl:write>
            </extensionActivity>
          </bpel:sequence>
        </bpel:flow>
      </bpel:while>
    </bpel:sequence>
    <bpel:sequence name="Sequence1">
      <extensionActivity>
        <xdl:read name="readPhotoMetadata">
          <template name="PhotoDetailsTemp"> photoMetadata </template>
          <data name="PhotoMetadata"> photoMetadataList </data>
        </xdl:read>
      </extensionActivity>
      <bpel:while name="While">
```

```
<bpel:flow name="Flow2">
    <extensionActivity>
        <xdl:read name="readPhotoFile">
            <template name="PhotoIDTemp"> photoID </template>
            <data name="PhotoFile"> photoFile </data>
        </xdl:read>
    </extensionActivity>
    <extensionActivity>
        <xdl:read name="readPhotoComment1">
            <template name="PhotoIDTemp"> photoID </template>
            <data name="PhotoFile"> photoComment </data>
        </xdl:read>
    </extensionActivity>
    <bpel:sequence name="Sequence">
        <extensionActivity>
            <xdl:read name="readPhotoComment2">
                <template name="PhotoIDTemp"> photoID </template>
                <data name="PhotoFile"> photoComment </data>
            </xdl:read>
        </extensionActivity>
        <extensionActivity>
            <xdl:write name="writePhotoComment">
                <data name="PhotoComment"> comment </data>
            </xdl:write>
        </extensionActivity>
    </bpel:sequence>
</bpel:flow>
</bpel:while>
</bpel:sequence>
</bpel:flow>
</bpel:if>
```

Bibliography

[1] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] G. Back. Datascript - a specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 66–77, London, UK, 2002. Springer-Verlag.

[4] D. Berardi, D. Calvanese, and G. De Giacomo. An industrial-strength description logic-based configurator platform. *Intelligent Systems and their Applications, IEEE*, 13:69 – 77, 1998.

[5] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on uml class diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005.

[6] A. Bertolino, G. Blair, F. Chauvel, C. F. Cortes, N. Georgantas, P. Grace, F. Howar, T. Huyn, B. Jonsson, M. Paolucci, A. Pathak, B. Souville, and M. Tivoli. Initial CONNECT architecture. Technical Report D1.1, CONNECT ICT FET IP Project, February 2010.

[7] S. Bloehdorn and A. Moschitti. Combined syntactic and semantic kernels for text classification. In *ECIR*, pages 307–318, 2007.

[8] Y.-D. Bromberg and V. Issarny. INDISS: Interoperable discovery system for networked services. In *Middleware*, pages 164–183, 2005.

[9] J. Cámara, C. Canal, and N. Vasilev. A framework for run-time behavioural service adaptation in ubiquitous computing. In *OTM Workshops*, pages 67–76, 2010.

[10] A. Carzaniga and A. Wolf. Content-based networking: A new communication infrastructure. *Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, 2002.

[11] D. Charlet, V. Issarny, and R. Chibout. Energy-efficient middleware-layer multi-radio networking: An assessment in the area of service discovery. *Computer Networks*, 52(1):4–24, 2008.

[12] S.-C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 8(1):49–78, 1999.

[13] CONNECT Consortium. Emergent connector for eternal software intensive networked systems. Fet proactive 6: Ict forever yours description of work, CONNECT, 2008.

[14] CONNECT Consortium. Compositional algebra of connectors. Technical Report D2.2, CONNECT, February 2011.

[15] CONNECT Consortium. Design of approaches for dependability and initial prototypes. Technical Report D5.2, CONNECT, February 2011.

[16] CONNECT Consortium. Experiment scenarios, prototypes and report iteration 1. Technical Report D6.2, CONNECT, February 2011.

[17] CONNECT Consortium. Further development of learning techniques. Technical Report D4.2, CONNECT, February 2011.

[18] CONNECT Consortium. Reasoning about and harmonizing the interaction behavior of networked systems at application- and middleware-layer. Technical Report D3.2, CONNECT, February 2011.

[19] P. P. da Silva, D. L. McGuinness, N. D. Rio, and L. Ding. Inference web in action: Lightweight use of the proof markup language. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, editors, *International Semantic Web Conference*, volume 5318 of *Lecture Notes in Computer Science*, pages 847–860. Springer, 2008.

[20] D. P. et al. Future internet: The cross ETP vision document. Technical report, http://www.future-internet.eu/fileadmin/documents/reports/Cross-ETPs_FI_Vision_Document_v1_0.pdf, 2009.

[21] J. Euzenat and P. Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007.

[22] J. Farrell and H. Lausen. Semantic annotations for wsdl and xml schema. In *http://www.w3.org/TR/sawsdl/*, August 2007.

[23] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 2–15, New York, NY, USA, 2006. ACM.

[24] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.

[25] H. Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, ICL, UK, 2006.

[26] A. Funk and K. Bontcheva. Ontology-based categorization of web services with machine learning. In *LREC*, 2010.

[27] Y. GIl. Description logics and planning. *AI Magazine, IEEE*, 24:73 – 84, 2005.

[28] P. Grace, G. S. Blair, and S. Samuel. Remmoc: A reflective middleware to support mobile client interoperability. In *CoopIS/DOA/ODBASE*, pages 1170–1187, 2003.

[29] D. Grigori, J. C. Corrales, and M. Bouzeghoub. Behavioral matchmaking for service retrieval. In *ICWS*, pages 145–152, 2006.

[30] GSMA. Rich communication suite, release 1f, 2010.

[31] E. Guttman, C. Perkins, and J. Veizades. Service location protocol – version 2, 1999.

[32] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. Wsmx - a semantic service-oriented architecture. In *ICWS*, pages 321–328, 2005.

[33] C. A. R. Hoare. Process algebra: A unifying approach. In *25 Years Comm. Seq. Processes*, 2004.

[34] M. Jeronimo and J. Weast. *UPnP design by example*. Intel Press, 2003.

[35] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[36] P. Kaplanski. Description logic as a common software engineering artifacts language. In *Proceedings of the 1st International Conference on Information Technology*. IEEE, 2008.

[37] M. Klusch, B. Fries, and K. P. Sycara. OWLS-MX: A hybrid semantic web service matchmaker for owl-s services. *J. Web Sem.*, 7(2):121–133, 2009.

[38] M. Klusch, P. Kapahnke, and I. Zinnikus. SAWSDL-MX2: A machine-learning approach for integrating semantic web service matchmaking variants. In *ICWS*, pages 335–342, 2009.

[39] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Softw., Pract. Exper.*, 32(2):135–164, 2002.

[40] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 101–114, New York, NY, USA, 2007. ACM.

[41] J. Magee and J. Kramer. *Concurrency : State models and Java programs*. Hoboken (N.J.) : Wiley, 2006.

[42] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. Owl-s: Semantic markup for web services. W3C Member Submission, 2004.

[43] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, M. Paolucci, K. P. Sycara, D. L. McGuinness, E. Sirin, and N. Srinivasan. Bringing semantics to web services with owl-s. In *World Wide Web*, pages 243–277, 2007.

[44] P. J. McCann and S. Chandra. Packet types: abstract specification of network protocol messages. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM.

[45] D. L. McGuinness. Configuration. In F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[46] Microsoft Press. Yahoo! and microsoft bridge global instant messaging communities, June 2010. online.

[47] S. Mokhtar, D. Preuveneers, N. Georgantas, V. Issarny, and Y. Berbers. EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support. *Journal of Systems and Software*, 81(5):785–808, 2008.

[48] S. B. Mokhtar, P.-G. Raverdy, A. Urbieta, and R. S. Cardoso. Interoperable semantic and syntactic service discovery for ambient computing environments. *IJACI*, 2(4):13–32, 2010.

[49] R. Monson-Haefel and D. Chappell. *Java message service*. O'Reilly Media, 2001.

[50] M. A. Motoyama and G. Varghese. Crosstalk: scalably interconnecting instant messaging networks. In *WOSN*, pages 61–68, 2009.

[51] D. Nau, M. Ghallab, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[52] O. Oanea, J. Sürmeli, and K. Wolf. Service discovery using communication fingerprints. In *ICSOC*, pages 612–618, 2010.

[53] OASIS. Univeral description, discovery and integration of web services. [online]. http://www.uddi.org.

[54] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.

[55] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *FASE*, pages 377–380, 2006.

[56] P. Raverdy, V. Issarny, R. Chibout, and A. de La Chapelle. A multi-protocol approach to service discovery and access in pervasive environments. In *Mobile and Ubiquitous Systems-Workshops, 2006. 3rd Annual International Conference on*, pages 1–9. IEEE, 2007.

[57] P.-G. Raverdy, O. Riva, A. de La Chapelle, R. Chibout, and V. Issarny. Efficient context-aware service discovery in multi-protocol pervasive environments. In *MDM*, page 3, 2006.

[58] F. Sailhan and V. Issarny. Scalable service discovery for manet. In *PerCom*, pages 235–244, 2005.

[59] P. Shvaiko, J. Euzenat, F. Giunchiglia, H. Stuckenschmidt, M. Mao, and I. Cruz, editors. *Proceedings of the 5th International Workshop on Ontology Matching (OM-2010)*. CEUR, 2010.

[60] J. Simmonds and M. C. Bastarrica. Description logics for consistency checking of architectural features in uml 2.0 models. Technical report, Department of Computer Science, University of Chile, 2000.

[61] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau. Htn planning for web service composition using shop2. *Journal of Web Semantics*, 1(4):377–396, 2004.

[62] R. Spalazzese, P. Inverardi, and V. Issarny. Towards a formalization of mediating connectors for on the fly interoperability. In *The IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA)*, 2010.

[63] D. Tantiprasut, J. Neil, and C. Farrell. Asn.1 protocol specification for use with arbitrary encoding schemes. *IEEE/ACM Trans. Netw.*, 5:502–513, August 1997.

[64] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[65] T. Vitvar, J. Kopecky, J. Viskova, and D. Fensel. Wsmo-lite annotations for web services. In M. Hauswirth, M. Koubarakis, and S. Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, Heidelberg, 2008. Springer Verlag.

[66] Y. Zhai, H. Su, and S. Zhan. A data flow optimization based approach for bpel processes partition. In *ICEBE*, pages 410–413, 2007.